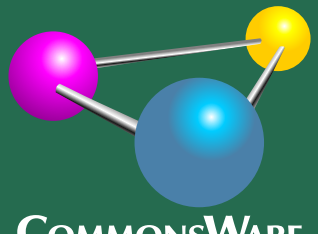
The Busy Coder's Guide to Android Development

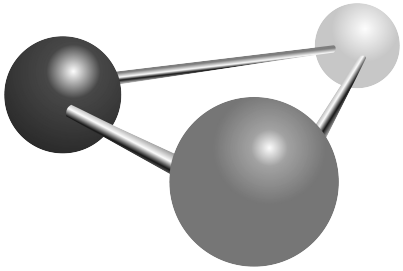
Mark L. Murphy



CommonsWare

The Busy Coder's Guide to Android Development

by Mark L. Murphy



Subscribe to updates at https://commonsware.com

Special Creative Commons BY-NC-SA 4.0 License Edition

The Busy Coder's Guide to Android Development
by Mark L. Murphy

Copyright © 2008-2014 CommonsWare, LLC. All Rights Reserved.
Printed in the United States of America.

Printing History:
January 2014: Version 6.4 ISBN: 978-0-9816780-0-9

The CommonsWare name and logo, "Busy Coder's Guide", and related trade dress are trademarks of CommonsWare, LLC.

All other trademarks referenced in this book are trademarks of their respective firms.

The publisher and author(s) assume no responsibility for errors or omissions or for damages resulting from the use of the information contained herein.

Table of Contents

Headings formatted in ***bold-italic*** have changed since the last version.

- Preface
  - Welcome to the Book! xxxv
  - The Book's Structure xxxv
  - ***The Trails ..... xxxvi***
  - About the Updates xli
  - Warescription xli
  - Book Bug Bounty xlii
  - Source Code and Its License xliii
  - ***Creative Commons and the Four-to-Free (42F) Guarantee .... xliv***
  - Acknowledgments xliv
- Key Android Concepts
  - Android Applications 1
  - Android Devices 7
  - Don't Be Scared 10
- Choosing Your Development Toolchain
  - Android Studio 11
  - Eclipse 11
  - IntelliJ IDEA 12
  - Command-Line Builds via Gradle for Android 12
  - Yet Other Alternatives 13
  - IDEs... And This Book 13
  - What We Are Not Covering 13
- Tutorial #1 - Installing the Tools
  - Step #1 - Checking Your Hardware Requirements 15
  - ***Step #2 - Setting Up Java and 32-Bit Linux Support 16***
  - Step #3 - Install the Developer Tools 16
  - ***Step #4 - Install the SDKs and Add-Ons 20***
  - In Our Next Episode... 28
- Android and Projects
  - ***Common Concepts 29***
  - Projects and Android Studio 30
  - Projects and Eclipse 37
  - Starter Project Generators 43
- Tutorial #2 - Creating a Stub Project
  - About Our Tutorial Project 45

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- About the Rest of the Tutorials 46
- About Our Tools 46
- Step #1: Creating the Project 47
- Step #2 - Set Up the Emulator 53
- Step #3 - Set Up the Device 65
- ***Step #4: Running the Project 68***
- ***In Our Next Episode... ..... 71***

- Getting Around Android Studio
  - Navigating The Project Explorer 73
  - Running Projects 76
  - Viewing Output 77
  - Accessing Android Tools 78
  - Android Studio and Release Channels 82
- Contents of Android Projects
  - ***What You Get, In General 85***
  - The Contents of an Android Studio Project 87
  - The Contents of an Eclipse Project 89
  - ***What You Get Out Of It 90***
- Introducing Gradle and the Manifest
  - Gradle: The Big Questions 91
  - Obtaining Gradle 94
  - Versions of Gradle and Gradle for Android 96
  - Gradle Environment Variables 97
  - ***Examining the Gradle Files 97***
  - ***Introducing the Manifest 99***
  - ***Things In Common Between the Manifest and Gradle 100***
  - ***Other Gradle Items of Note 103***
  - ***The Rest of the Manifest 105***
  - ***Learning More About Gradle 107***
  - Visit the Trails! 107
- Tutorial #3 - Changing Our Manifest (and Gradle File)
  - Some Notes About Relative Paths 109
  - Step #1: Supporting Screens 110
  - Step #2: Adding our Minimum and Target SDK Versions 113
  - In Our Next Episode... 115
- Some Words About Resources
  - String Theory 118
  - Got the Picture? 122
  - Dimensions 126
  - The Resource That Shall Not Be Named... Yet 128
- Tutorial #4 - Adjusting Our Resources

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - Step #1: Changing the Name 129
    - Step #2: Changing the Icon 131
    - Step #3: Running the Result 140
    - In Our Next Episode... 142
- The Theory of Widgets
    - What Are Widgets? 143
    - Size, Margins, and Padding 145
    - What Are Containers? 145
    - The Absolute Positioning Anti-Pattern 146
    - ***The Theme of This Section: Themes 147***
- The Android User Interface
    - ***The Activity 151***
    - Dissecting the Activity 152
    - Using XML-Based Layouts 153
- Basic Widgets
    - Common Concepts 159
    - Assigning Labels 162
    - A Commanding Button 172
    - Fleeting Images 177
    - Fields of Green. Or Other Colors. 184
    - More Common Concepts 190
    - Visit the Trails! 192
- Debugging Your App
    - Get Thee To a Stack Trace 194
    - The Case of the Confounding Class Cast 198
    - Point Break 199
- LinearLayout and the Box Model
    - Concepts and Properties 201
    - Android Studio Graphical Layout Editor 205
    - Eclipse Graphical Layout Editor 206
- Other Common Widgets and Containers
    - Just a Box to Check 209
    - Don't Like Checkboxes? How About Toggles or Switches? 214
    - Turn the Radio Up 219
    - ***All Things Are Relative 223***
    - Tabula Rasa 230
    - Scrollwork 237
    - Making Progress with ProgressBars 240
    - Visit the Trails! 241
- Tutorial #5 - Making Progress
    - Step #1: Removing The "Hello, World" 243

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - Step #2: Adding a ProgressBar 245
    - Step #3: Seeing the Results 248
    - In Our Next Episode... 248
- GUI Building, Continued
    - Making Your Selection 249
    - Including Includes 250
    - Wrap It Up (In a Container) 252
    - Morphing Widgets 253
    - Preview of Coming Attractions 254
- AdapterViews and Adapters
    - Adapting to the Circumstances 255
    - Lists of Naughty and Nice 257
    - Clicks versus Selections 259
    - Spin Control 263
    - Grid Your Lions (Or Something Like That...) 266
    - Fields: Now With 35% Less Typing! 269
    - Customizing the Adapter 273
    - Visit the Trails! 281
- The WebView Widget
    - Role of WebView 283
    - WebView and WebKit 284
    - Adding the Widget 284
    - Loading Content Via a URL 285
    - Supporting JavaScript 286
    - Alternatives for Loading Content 287
    - ***Listening for Events 288***
    - WebView and Android 4.4 291
    - Visit the Trails! 292
- Defining and Using Styles
    - Styles: DIY DRY 293
    - Elements of Style 295
    - Themes: Would a Style By Any Other Name... 298
    - What Happens If You Have No Theme 298
- JARs and Library Projects
    - The Dalvik VM, and a Bit of ART 302
    - Getting the Library 302
    - The Outer Limits 304
    - JAR Dependency Management 305
    - OK, So What is a Library Project? 305
    - Using a Library Project 306
    - Library Projects: What You Get 307

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- The Android Support Package 308
- Tutorial #6 - Adding a Library
  - Step #1: Attaching the Android Support Package 313
  - Step #2: Downloading the Third-Party JARs 314
  - In Our Next Episode... 316
- The Action Bar
  - Bar Hopping 317
  - Yet Another History Lesson 323
  - Your Action Bar Options 324
  - ***Setting the Target 325***
  - Defining the Resource 327
  - Applying the Resource 329
  - Responding to Events 330
  - The Rest of the Sample Activity 331
  - MENU Key, We Hardly Knew Ye 337
  - ***Action Bars, Live in Living Color! 338***
  - Visit the Trails! 348
- Tutorial #7 - Setting Up the Action Bar
  - Step #1: Acquiring Some Icons 349
  - Step #2: Defining Some Options 350
  - ***Step #3: Loading and Responding to Our Options 353***
  - In Our Next Episode... 357
- Android's Process Model
  - When Processes Are Created 359
  - BACK, HOME, and Your Process 360
  - Termination 361
  - Foreground Means "I Love You" 362
  - You and Your Heap 362
- Activities and Their Lifecycles
  - Creating Your Second (and Third and...) Activity 364
  - Warning! Contains Explicit Intents! 370
  - Using Implicit Intents 372
  - Extra! Extra! 377
  - Pondering Parcelable 379
  - Asynchronicity and Results 380
  - Schroedinger's Activity 380
  - Life, Death, and Your Activity 381
  - When Activities Die 383
  - Walking Through the Lifecycle 384
  - Recycling Activities 387
  - Application: Transcending the Activity 388

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Tutorial #8 - Setting Up An Activity
  - Step #1: Creating the Stub Activity Class and Manifest Entry 391
  - Step #2: Launching Our Activity 394
  - In Our Next Episode... 395
- The Tactics of Fragments
  - The Six Questions 397
  - Where You Get Your Fragments From 400
  - Your First Fragment 400
  - The Fragment Lifecycle Methods 403
  - Your First Dynamic Fragment 404
  - Fragments and the Action Bar 407
  - Fragments Within Fragments: Just Say "Maybe" 409
  - Fragments and Multiple Activities 409
- Tutorial #9 - Starting Our Fragments
  - Step #1: Create a SimpleContentFragment 411
  - Step #2: Examining SimpleContentFragment 414
  - In Our Next Episode... 414
- Swiping with ViewPager
  - Swiping Design Patterns 415
  - Pieces of a Pager 416
  - Paging Fragments 416
  - Paging Other Stuff 421
  - Indicators 422
- Tutorial #10 - Rigging Up a ViewPager
  - Step #1: Add a ViewPager to the Layout 427
  - Step #2: Obtaining Our ViewPager 428
  - Step #3: Creating a ContentsAdapter 429
  - ***Step #4: Setting Up the ViewPager 430***
  - In Our Next Episode... 431
- Resource Sets and Configurations
  - What's a Configuration? And How Do They Change? 433
  - Configurations and Resource Sets 434
  - Screen Size and Orientation 435
  - Coping with Complexity 438
  - Choosing The Right Resource 439
  - API-Versioned Resources 443
  - Default Change Behavior 445
  - State Saving Scenarios 446
  - Your Options for Configuration Changes 447
  - ***Blocking Rotations 459***
  - And Now, a Word From the Android Project View 459

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Material Design Basics
  - Your App, in Technicolor! 463
- Dealing with Threads
  - The Main Application Thread 471
  - Getting to the Background 473
  - Asyncing Feeling 473
  - Alternatives to AsyncTask 483
  - And Now, The Caveats 484
  - Event Buses 485
  - Visit the Trails! 492
- Requesting Permissions
  - Mother, May I? 494
  - New Permissions in Old Applications 496
  - Permissions: Up Front Or Not At All 496
  - Signature Permissions 497
  - Requiring Permissions 498
- Assets, Files, and Data Parsing
  - Packaging Files with Your App 499
  - Files and Android 501
  - Working with Internal Storage 502
  - Working with External Storage 505
  - Multiple User Accounts 509
  - Linux Filesystems: You Sync, You Win 510
  - StrictMode: Avoiding Janky Code 511
  - XML Parsing Options 514
  - JSON Parsing Options 515
  - Visit the Trails! 515
- Tutorial #11 - Adding Simple Content
  - Step #1: Adding Some Content 517
  - Step #2: Using SimpleContentFragment 518
  - Step #3: Launching Our Activities, For Real This Time 519
  - In Our Next Episode... 521
- Tutorial #12 - Displaying the Book
  - Step #1: Adding a Book 523
  - Step #2: Creating a ModelFragment 524
  - Step #3: Defining Our Model 525
  - Step #4: Examining Our Model 527
  - Step #5: Defining Our Event 527
  - Step #6: Loading Our Model 529
  - Step #7: Registering for Events 531
  - Step #8: Adapting the Content 532

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- ***Step #9: Showing the Content When Loaded 533***
- Step #10: Attaching our ModelFragment 533
- ***Step #11: Showing the Content After a Configuration Change 535***
- Step #12: Going Home, Again 535
- Step #13: Setting Up StrictMode 536
- In Our Next Episode... 537

- Using Preferences
  - Getting What You Want 539
  - Stating Your Preference 540
  - Introducing PreferenceActivity 541
  - Types of Preferences 551
  - Intents for Headers or Preferences 554
  - Conditional Headers 555
- Tutorial #13 - Using Some Preferences
  - Step #1: Defining the Preference XML Files 561
  - Step #2: Creating Our PreferenceActivity 564
  - Step #3: Adding To Our Action Bar 566
  - Step #4: Launching the PreferenceActivity 567
  - ***Step #5: Loading the Preferences 571***
  - Step #6: Saving the Last-Read Position 574
  - Step #7: Restoring the Last-Read Position 576
  - ***Step #9: Keeping the Screen On 576***
  - In Our Next Episode... 578
- SQLite Databases
  - Introducing SQLite 579
  - Thinking About Schemas 580
  - Start with a Helper 580
  - ***Getting Data Out 585***
  - The Rest of the CRUD 591
  - Hey, What About Hibernate? 597
  - Visit the Trails! 597
- Tutorial #14 - Saving Notes
  - Step #1: Adding a DatabaseHelper 599
  - Step #2: Examining DatabaseHelper 601
  - Step #3: Creating a NoteFragment 601
  - Step #4: Examining NoteFragment 606
  - Step #5: Creating the NoteActivity 606
  - Step #6: Examining NoteActivity 608
  - Step #7: Add Notes to the Action Bar 608
  - Step #8: Defining a NoteLoadedEvent 610
  - Step #9: Loading a Note from the Database 611

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Step #10: Loading the Note Into the Fragment 612
- Step #11: Updating the Database 613
- Step #12: Saving the Note 614
- Step #13: Adding a Delete Action Bar Item 616
- Step #14: Closing the NoteFragment When Deleted 618
- In Our Next Episode... 623

- Internet Access
  - DIY HTTP 625
  - HTTP via DownloadManager 635
  - Using Third-Party JARs 636
  - SSL 637
  - Using HTTP Client Libraries 637
  - Visit the Trails 651
- Intents, Intent Filters
  - What's Your Intent? 653
  - Stating Your Intent(ions) 655
  - Responding to Implicit Intents 655
  - ***Requesting Implicit Intents 658***
  - ShareActionProvider 662
- Broadcasts and Broadcast Receivers
  - The Stopped State 667
  - Example System Broadcasts 668
  - The Order of Things 676
  - Keeping It Local 677
  - Visit the Trails! 678
- Tutorial #15 - Sharing Your Notes
  - Step #1: Adding a ShareActionProvider 679
  - Step #2: Sharing the Note 680
  - Step #3: Testing the Result 682
  - In Our Next Episode... 683
- Services and the Command Pattern
  - Why Services? 685
  - Setting Up a Service 686
  - Communicating To Services 688
  - Scenario: The Music Player 690
  - Communicating From Services 693
  - Scenario: The Downloader 695
- Tutorial #16 - Updating the Book
  - Step #1: Adding a Stub DownloadCheckService 701
  - Step #2: Tying the Service Into the Action Bar 703
  - Step #3: Defining Our Event 704

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Step #4: Defining Our JSON 705
- Step #5: Defining Our Retrofit Interface 706
- Step #6: Retrieving Our JSON Via Retrofit 707
- Step #7: Downloading the Update 708
- Step #8: Unpacking the Update 710
- Step #9: Using the Update 714
- In Our Next Episode... 719

- AlarmManager and the Scheduled Service Pattern
  - Scenarios 721
  - Options 722
  - A Simple Example 724
  - The Five set...() Varieties 726
  - The Four Types of Alarms 727
  - When to Schedule Alarms 728
  - Archetype: Scheduled Service Polling 730
  - Staying Awake at Work 733
  - Warning: Not All Android Devices Play Nice 737
  - Debugging Alarms 738
  - WakefulBroadcastReceiver 741
- Tutorial #17 - Periodic Book Updates
  - Step #1: Adding a Stub UpdateReceiver 745
  - Step #2: Scheduling the Alarms 747
  - Step #3: Adding the WakefulIntentService 748
  - Step #4: Using WakefulIntentService 750
  - ***Step #5: Completing the UpdateReceiver 750***
  - In Our Next Episode... 751
- Notifications
  - What's a Notification? 753
  - Showing a Simple Notification 756
  - The Activity-Or-Notification Scenario 761
  - Big (and Rich) Notifications 761
  - ***Foreground Services 768***
  - Disabled Notifications 771
- Tutorial #18 - Notifying the User
  - ***Step #1: Raising the Notification 775***
  - In Our Next Episode... 777
- Large-Screen Strategies and Tactics
  - Objective: Maximum Gain, Minimum Pain 779
  - The Fragment Strategy 779
  - Fragment Example: The List-and-Detail Pattern 788
  - Other Master-Detail Strategies 800

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Showing More Pages 812
- Fragment FAQs 818
- Screen Size and Density Tactics 819
- ***Other Considerations 822***

- Tutorial #19 - Supporting Large Screens
  - Step #1: Creating Our Layouts 827
  - Step #2: Loading Our Sidebar Widgets 833
  - Step #3: Opening the Sidebar 834
  - Step #4: Loading Content Into the Sidebar 835
  - ***Step #5: Removing Content From the Sidebar 837***
- Backwards Compatibility Strategies and Tactics
  - Think Forwards, Not Backwards 845
  - Aim Where You Are Going 847
  - A Target-Rich Environment 847
  - Lint: It's Not Just For Belly Buttons 848
  - A Little Help From Your Friends 849
  - Avoid the New on the Old 849
  - Testing 853
  - Keeping Track of Changes 853
- Getting Help
  - Questions. Sometimes, With Answers. 855
  - Heading to the Source 856
  - Getting Your News Fix 857
- Working with Library Projects
  - Prerequisites 859
  - Creating a Library Project 859
  - ***Using a Library Project, Part II 863***
  - Library Projects and the Manifest 863
  - Limitations of Library Projects 864
- Gradle and Eclipse Projects
  - Prerequisites and Warnings 865
  - "Legacy"? 865
  - ***Creating Your Gradle Build File 866***
  - Examining the Gradle File 871
- Gradle and Tasks
  - ***Key Build-Related Tasks 873***
  - Results 875
- Gradle and the New Project Structure
  - Prerequisites and Warnings 877
  - Objectives of the New Project Structure 878
  - Terminology 878

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- ***Creating a Project in the New Structure 882***
- What the New Project Structure Looks Like 882
- ***Configuring the Stock Build Types 885***
- Adding Build Types 890
- Adding Product Flavors and Getting Build Variants 892
- Doing the Splits 896
- Revisiting the Legacy Gradle File 898
- Working with the New Project Structure in Android Studio 900

- Gradle and Dependencies
  - Prerequisites and Warnings 903
  - "Dependencies"? 904
  - ***A Tale of Two Dependencies Closures 904***
  - ***Depending Upon a JAR 904***
  - Depending Upon NDK Binaries 906
  - ***Depending Upon an Android Library Project 906***
  - Depending Upon Sub-Projects 908
  - ***Depending Upon Artifacts 909***
  - Creating Android JARs from Gradle 917
  - A Property of Transitive (Dependencies) 918
  - Dependencies By Build Type 918
  - Dependencies By Flavor 919
  - ***Examining Some CWAC Builds 920***
- Manifest Merger Rules
  - Prerequisites 927
  - Manifest Scenarios 928
  - Pieces of Manifest Generation 929
  - Examining the Merger Results 931
  - Merging Elements and Attributes 932
  - Employing Placeholders 938
- Signing Your App
  - Prerequisites 943
  - Role of Code Signing 943
  - What Happens In Debug Mode 944
  - ***Production Signing Keys 945***
- Distribution
  - Prerequisites 961
  - ***Get Ready To Go To Market 961***
- Advanced Gradle for Android Tips
  - Prerequisites 967
  - ***Gradle, DRY 967***
  - ***Automating APK Version Information 973***

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Adding to BuildConfig 975
- Down and Dirty with the DSL 977

- JUnit and Android
  - ***Prerequisites 981***
  - JUnit Basics 981
  - Pondering Gradle for Android 982
  - Where Your Test Code Lives 983
  - Where Your Test Code Runs 987
  - Writing Your Test Cases 988
  - Your Test Suite 993
  - Running Your Tests 994
  - Testing Android Library Projects 1000
  - Test Dependencies 1001
  - Testing Legacy Project Structures with Gradle for Android 1002
- Testing with JUnit4
  - ***Prerequisites 1005***
  - ***The AndroidJUnitRunner 1005***
  - ***JUnit4 ..... 1008***
- MonkeyRunner and the Test Monkey
  - Prerequisites 1017
  - MonkeyRunner 1017
  - Monkeying Around 1019
- Testing with UIAutomator
  - Prerequisites 1021
  - What Is UIAutomator? 1021
  - Why Choose UIAutomator Over Alternatives? 1022
  - Creating Some Tests 1022
  - Running Your Tests 1031
  - Finding Your Widgets 1032
- Introducing GridLayout
  - Prerequisites 1035
  - Issues with the Classic Containers 1035
  - The New Contender: GridLayout 1037
  - ***GridLayout and the Android Support Package 1037***
  - Eclipse and GridLayout 1039
  - Trying to Have Some Rhythm 1039
  - Our Test App 1040
  - Replacing the Classics 1042
  - Implicit Rows and Columns 1048
  - Row and Column Spans 1050
- Dialogs and DialogFragments

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - Prerequisites 1055
    - DatePickerDialog and TimePickerDialog 1055
    - AlertDialog 1061
    - DialogFragments 1062
    - DialogFragment: The Other Flavor 1066
    - Dialogs: Modal, Not Blocking 1067
- Advanced ListViews
    - Prerequisites 1069
    - Multiple Row Types, and Self Inflation 1069
    - Choice Modes and the Activated Style 1075
    - Custom Mutable Row Contents 1076
    - From Head To Toe 1082
- Action Bar Navigation
    - Prerequisites 1087
    - List Navigation 1087
    - Tabs (And Sometimes List) Navigation 1092
    - Custom Navigation 1098
- Action Modes and Context Menus
    - Prerequisites 1100
    - Another Wee Spot O' History 1100
    - Manual Action Modes 1101
    - Multiple-Choice-Modal Action Modes 1105
    - Long-Click To Initiate an Action Mode 1109
    - Split Action Modes 1114
- Other Advanced Action Bar Techniques
    - Prerequisites 1119
    - Splitting the Bar 1119
    - Floating Action Bars 1123
    - Action Layouts, Action Views, and Action Providers 1125
    - Searching with SearchView 1126
- AppCompat: The Official Action Bar Backport
    - Prerequisites 1133
    - Ummmm... Why? 1133
    - The Basics of Using AppCompat 1135
    - Other AppCompat Effects 1142
    - And Then, There Are the Bugs 1148
    - To Material, or Not to Material 1149
- ActionBarSherlock
    - Prerequisites 1153
    - Installation 1154
    - Code Changes 1154

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Implementing a Navigation Drawer
  - Prerequisites 1157
  - What is a Navigation Drawer? 1157
  - A Simple Navigation Drawer 1159
  - Alternative Row Layouts 1165
  - Additional Considerations 1167
  - What Should Not Be in the Drawer 1176
  - Independent Implementations 1176
- Advanced Uses of WebView
  - Prerequisites 1179
  - Friends with Benefits 1179
  - Turnabout is Fair Play 1185
  - Navigating the Waters 1189
  - Settings, Preferences, and Options (Oh, My!) 1190
  - Security and Your WebView 1190
- The Input Method Framework
  - Prerequisites 1195
  - Keyboards, Hard and Soft 1195
  - Tailored To Your Needs 1196
  - Tell Android Where It Can Go 1201
  - Fitting In 1203
  - Jane, Stop This Crazy Thing! 1205
- Fonts
  - Prerequisites 1207
  - Love The One You're With 1207
  - Yeah, But Do We Really Have To Do This in Java? 1211
  - Here a Glyph, There a Glyph 1212
- Rich Text
  - Prerequisites 1215
  - The Span Concept 1215
  - Loading Rich Text 1217
  - Editing Rich Text 1219
  - Saving Rich Text 1224
  - Manipulating Rich Text 1225
- Animators
  - Prerequisites 1227
  - ViewPropertyAnimator 1227
  - The Foundation: Value and Object Animators 1232
  - Animating Custom Types 1235
  - Hardware Acceleration 1236
  - The Three-Fragment Problem 1237

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Legacy Animations
  - Prerequisites 1249
  - It's Not Just For Toons Anymore 1249
  - A Quirky Translation 1250
  - Fading To Black. Or Some Other Color. 1254
  - When It's All Said And Done 1256
  - Loose Fill 1257
  - Hit The Accelerator 1257
  - Animate. Set. Match. 1258
  - Active Animations 1259
- Custom Drawables
  - Prerequisites 1261
  - ColorDrawable 1262
  - AnimationDrawable 1262
  - StateListDrawable 1265
  - ColorStateList 1267
  - LayerDrawable 1268
  - TransitionDrawable 1270
  - LevelListDrawable 1270
  - ScaleDrawable and ClipDrawable 1272
  - InsetDrawable 1281
  - ShapeDrawable 1282
  - BitmapDrawable 1292
  - Composite Drawables 1299
  - A Stitch In Time Saves Nine 1303
- Mapping with Maps V2
  - Prerequisites 1313
  - A Brief History of Mapping on Android 1314
  - Where You Can Use Maps V2 1315
  - Licensing Terms for Maps V2 1315
  - What You Need to Start 1316
  - The Book Samples... And You! 1320
  - Setting Up a Basic Map 1320
  - Playing with the Map 1325
  - Map Tiles 1328
  - Placing Simple Markers 1328
  - Seeing All the Markers 1331
  - Flattening and Rotating Markers 1333
  - Sprucing Up Your "Info Windows" 1337
  - Images and Your Info Window 1342
  - Setting the Marker Icon 1348

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Responding to Taps 1349
- Dragging Markers 1351
- The “Final” Limitations 1353
- A Bit More About IPC 1356
- Finding the User 1357
- Drawing Lines and Areas 1361
- Gestures and Controls 1364
- Tracking Camera Changes 1365
- Maps in Fragments and Pagers 1367
- Animating Marker Movement 1372
- Maps, of the Indoor Variety 1381
- Taking a Snapshot of a Map 1381
- MapFragment vs. MapView 1382
- About That AbstractMapActivity Class... 1383
- Helper Libraries for Maps V2 1387
- Problems with Maps V2 at Runtime 1391
- Problems with Maps V2 Deployment 1391
- What Non-Compliant Devices Show 1391
- Mapping Alternatives 1392
- News and Getting Help 1392

• Crafting Your Own Views
- Prerequisites 1395
- Pick Your Poison 1395
- Colors, Mixed How You Like Them 1396
- ReverseChronometer: Simply a Custom Subclass 1407
- AspectLockedFrameLayout: A Custom Container 1412
- Mirror and MirroringFrameLayout: Draw It Yourself 1416

• Custom Dialogs and Preferences
- Prerequisites 1427
- Your Dialog, Chocolate-Covered 1427
- Preferring Your Own Preferences, Preferably 1431

• Progress Indicators
- Prerequisites 1439
- Progress Bars 1439
- ProgressBar and Threads 1442
- Tailoring Progress Bars 1445
- Progress Dialogs 1453
- Title Bar and Action Bar Progress Indicators 1455
- Direct Progress Indication 1457

• Advanced Notifications
- Prerequisites 1459

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - Being a Good Citizen 1459
    - Wear? There! 1460
    - Stacking Notifications 1465
    - Avoiding Wear 1471
    - Other Wear-Specific Notification Options 1472
    - ***Lockscreen Notifications 1485***
    - Priority, and Heads-Up Notifications 1494
    - Full-Screen Notifications 1496
    - Custom Views: or How Those Progress Bars Work (Sometimes) 1499
    - Seeing It In Action 1500
    - How You Really Do Progress Notifications 1506
    - Life After Delete 1509
    - The Mysterious Case of the Missing Number 1510
- More Fun with Pagers
    - Prerequisites 1511
    - Hosting ViewPager in a Fragment 1511
    - Pages and the Action Bar 1513
    - ViewPagers and Scrollable Contents 1515
    - Using ViewPagerIndicator 1516
    - Columns for Large, Pages for Small 1520
    - Introducing ArrayPagerAdapter 1526
    - Columns for Large Landscape, Pages for the Rest 1529
    - Adding, Removing, and Moving Pages 1534
    - Inside ArrayPagerAdapter 1538
- Focus Management and Accessibility
    - Prerequisites 1551
    - ***Prepping for Testing 1552***
    - Controlling the Focus 1552
    - Accessibility and Focus 1561
    - Accessibility Beyond Focus 1562
    - Accessibility Beyond Impairment 1572
- Miscellaneous UI Tricks
    - Prerequisites 1575
    - Full-Screen and Lights-Out Modes 1575
    - Offering a Delayed Timeout 1586
- Event Buses Alternatives
    - Prerequisites 1591
    - A Brief Note About the Sample Apps 1591
    - Standard Intents as Event Bus 1591
    - LocalBroadcastManager as Event Bus 1592
    - Square's Otto 1602

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - Revisiting greenrobot's EventBus 1608
- Home Screen App Widgets
    - Prerequisites 1615
    - East is East, and West is West... 1616
    - The Big Picture for a Small App Widget 1616
    - Crafting App Widgets 1617
    - Another and Another 1624
    - App Widgets: Their Life and Times 1625
    - Controlling Your (App Widget's) Destiny 1625
    - Change Your Look 1626
    - One Size May Not Fit All 1627
    - Lockscreen Widgets 1634
    - Preview Images 1640
    - Being a Good Host 1642
- Adapter-Based App Widgets
    - Prerequisites 1643
    - AdapterViews for App Widgets 1643
    - Building Adapter-Based App Widgets 1644
- Content Provider Theory
    - Prerequisites 1659
    - Using a Content Provider 1659
    - Building Content Providers 1666
    - Issues with Content Providers 1673
- Content Provider Implementation Patterns
    - Prerequisites 1675
    - The Single-Table Database-Backed Content Provider 1675
    - The Local-File Content Provider 1683
    - The Protected Provider 1690
    - The Stream Provider 1693
    - FileProvider 1696
    - StreamProvider 1700
- The Loader Framework
    - Prerequisites 1703
    - Cursors: Issues with Management 1704
    - Introducing the Loader Framework 1704
    - Honeycomb... Or Not 1706
    - Using CursorLoader 1706
    - What Else Is Missing? 1708
    - Issues, Issues, Issues 1709
    - Loaders Beyond Cursors 1709
    - What Happens When...? 1709

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- The ContactsContract Provider
  - Prerequisites 1713
  - Introducing You to Your Contacts 1714
  - Pick a Peck of Pickled People 1715
  - Spin Through Your Contacts 1718
  - Makin' Contacts 1727
- The CalendarContract Provider
  - Prerequisites 1734
  - You Can't Be a Faker 1734
  - Do You Have Room on Your Calendar? 1734
  - Penciling In an Event 1739
- The MediaStore Provider
  - Prerequisites 1741
  - What Is the MediaStore? 1742
  - MediaStore and "Other" External Storage 1743
  - How Does My Content Get Indexed? 1744
  - ***How Do I Retrieve Video from the MediaStore? 1744***
- Consuming Documents
  - Prerequisites 1753
  - The Storage Access... What? 1753
  - The Storage Access Framework Participants 1755
  - Picking How to Pick (a Peck of Pickled Pepper Photos) 1755
  - Opening a Document 1756
  - The Rest of the CRUD 1759
  - Pondering Persistent Permissions 1760
- Providing Documents
  - Prerequisites 1763
  - Have Your Content, and Provide it Too 1763
  - Key Provider Concepts 1765
  - Pieces of a Provider 1766
  - Optional Provider Capabilities 1780
- Encrypted Storage
  - Prerequisites 1788
  - Scenarios for Encryption 1788
  - Obtaining SQLCipher 1789
  - Attaching SQLCipher To Your Project 1789
  - Using SQLCipher 1790
  - SQLCipher Limitations 1793
  - Passwords and Sessions 1794
  - About Those Passphrases... 1794
  - Encrypted Preferences 1802

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- IOCipher 1804
- Packaging and Distributing Data
  - Prerequisites 1805
  - ***Packing a Database To Go 1805***
- Advanced Database Techniques
  - Prerequisites 1809
  - Full-Text Indexing 1809
- Miscellaneous Network Capabilities
  - Prerequisites 1825
  - Downloading Files 1825
- Audio Playback
  - Prerequisites 1839
  - Get Your Media On 1839
  - MediaPlayer for Audio 1840
  - Other Ways to Make Noise 1846
- Audio Recording
  - Prerequisites 1849
  - Recording by Intent 1849
  - Recording to Files 1852
  - Recording to Streams 1855
  - Raw Audio Input 1858
  - ***Requesting the Microphone 1858***
- Video Playback
  - Prerequisites 1861
  - Moving Pictures 1861
- Using the Camera via 3rd-Party Apps
  - Prerequisites 1867
  - Being Specific About Features 1867
  - Still Photos: Letting the Camera App Do It 1868
  - Scanning with ZXing 1870
  - Videos: Letting the Camera App Do It 1871
  - Directly Working with the Camera 1873
- Working Directly with the Camera
  - Prerequisites 1876
  - Basic CameraFragment Usage 1876
  - Simple Configuration and Usage 1877
  - Core Camera Concepts 1884
  - Advanced CWAC-Camera Features 1907
- Media Routes
  - Prerequisites 1915
  - Terminology 1915

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - ***A Tale of Three MediaRouters 1916***
    - Attaching to MediaRouter 1917
    - ***User Route Selection with MediaRouteActionProvider 1919***
    - Using Live Audio Routes 1936
    - Using Live Video Routes 1936
    - ***Using Remote Playback Routes 1936***
- Supporting External Displays
    - Prerequisites 1955
    - ***A History of External Displays 1955***
    - What is a Presentation? 1956
    - Playing with External Displays 1957
    - Detecting Displays 1963
    - A Simple Presentation 1964
    - A Simpler Presentation 1969
    - Presentations and Configuration Changes 1974
    - Presentations as Fragments 1975
    - Another Sample Project: Slides 1985
    - Device Support for Presentation 1992
    - Presentations from a Service 1993
    - Hey, What About Chromecast? 1996
- Google Cast and Chromecast
    - Prerequisites 1997
    - Here a Cast, There a Cast 1997
    - Common Chromecast Development Notes 1999
    - Your API Choices 1999
    - Senders and Receivers 2000
    - Supported Media Types 2001
    - Cast SDK Dependencies 2002
    - Developing Google Cast Apps 2004
- The “Ten-Foot UI”
    - ***Prerequisites 2005***
    - What is the “Ten-Foot UI”? 2006
    - ***Overscan 2006***
    - ***Navigation 2007***
    - ***Stylistic Considerations 2008***
    - ***The Leanback UI 2010***
    - ***Testing Your Theories 2024***
- Putting the TVs All Together: Decktastic
    - ***Prerequisites 2028***
    - ***Introducing Decktastic 2028***
    - ***Trying Decktastic Yourself 2032***

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- ***Implementing Decktastic 2032***

- Creating a MediaRouteProvider
  - Prerequisites 2065
  - Terminology 2065
  - DIY Chromecast 2066
  - Creating the MediaRouteProvider 2068
  - Consuming the MediaRouteProvider 2078
  - Implementing This "For Realz" 2081
- SSL
  - Prerequisites 2085
  - Basic SSL Operation 2085
  - Common SSL Problems 2086
  - TrustManagers 2087
  - TrustManagerBuilder 2090
  - About That Man in the Middle 2093
  - Self-Signed Certificates, Revisited 2095
  - Certificate Memorizing 2096
  - Pinning 2100
  - NetCipher 2101
- Advanced Permissions
  - Prerequisites 2103
  - Securing Yourself 2103
  - Signature Permissions 2106
  - The Custom Permission Vulnerability 2108
- Restricted Profiles and UserManager
  - Prerequisites 2119
  - Android Tablets and Multiple User Accounts 2119
  - Determining What the User Can Do 2125
  - Impacts of Device-Level Restrictions 2128
  - Enabling Custom Restrictions 2128
  - Implicit Intents May Go "Boom" 2139
  - The Future: App Ops? 2139
- Tapjacking
  - Prerequisites 2141
  - What is Tapjacking? 2141
  - Detecting Potential Tapjackers 2146
  - ***Defending Against Tapjackers 2148***
  - Why Is This Being Discussed? 2151
  - What Changed in 4.0.3? 2152
- Miscellaneous Security Techniques
  - Prerequisites 2153

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - Public Key Validation 2153
    - Choosing Your Signing Keysize 2169
    - Avoiding Accidental APIs 2170
    - Other Ways to Expose Data 2175
- Accessing Location-Based Services
    - Prerequisites 2179
    - Location Providers: They Know Where You're Hiding 2180
    - Finding Yourself 2180
    - On the Move 2182
    - Are We There Yet? Are We There Yet? Are We There Yet? 2183
    - Testing... Testing... 2184
    - Alternative Flavors of Updates 2185
    - The Fused Option 2186
- The Fused Location Provider
    - Prerequisites 2187
    - Why Use the Fused Location Provider? 2187
    - Why Not Use the Fused Location Provider? 2188
    - Finding Our Location, Once 2188
    - Requesting Location Updates 2197
    - Gaps in the Fused Location Provider 2199
- Working with the Clipboard
    - Prerequisites 2201
    - Using the Clipboard on Android 1.x/2.x 2201
    - Advanced Clipboard on Android 3.x and Higher 2204
    - Monitoring the Clipboard 2209
    - The Android 4.3 Clipboard Bug 2211
- Telephony
    - Prerequisites 2213
    - Report To The Manager 2214
    - You Make the Call! 2214
    - No, Really, You Make the Call! 2217
- Working With SMS
    - Prerequisites 2219
    - ***Sending Out an SOS, Give or Take a Letter 2220***
    - Monitoring and Receiving SMS 2227
    - The SMS Inbox 2233
    - Asking to Change the Default 2234
    - SMS and the Emulator 2235
- NFC
    - Prerequisites 2237
    - What Is NFC? 2237

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - To NDEF, Or Not to NDEF 2239
    - NDEF Modalities 2239
    - NDEF Structure and Android's Translation 2240
    - The Reality of NDEF 2241
    - Sources of Tags 2243
    - Writing to a Tag 2243
    - Responding to a Tag 2251
    - Expected Pattern: Bootstrap 2252
    - Mobile Devices are Mobile 2253
    - Enabled and Disabled 2253
    - Android Beam 2254
    - Beaming Files 2260
    - Another Sample: SecretAgentMan 2262
    - Additional Resources 2271
- Device Administration
    - Prerequisites 2273
    - Objectives and Scope 2273
    - Defining and Registering an Admin Component 2274
    - Going Into Lockdown 2280
    - Passwords and Device Administration 2287
    - Getting Along with Others 2291
- PowerManager and WakeLocks
    - Prerequisites 2293
    - Keeping the Screen On, UI-Style 2293
    - The Role of the WakeLock 2294
    - What WakefulIntentService Does 2295
- JobScheduler
    - Prerequisites 2297
    - The Limitations of AlarmManager 2297
    - Enter the JobScheduler 2298
    - Employing JobScheduler 2298
    - Pondering Backoff Criteria 2311
    - Other JobScheduler Features 2312
- Push Notifications with GCM
    - Prerequisites 2315
    - The Precursor: C2DM 2316
    - The Replacement: GCM 2316
    - The Re-Replacement: GCM 2013 2316
    - The Pieces of Push 2317
    - ***A Simple Push 2323***
    - Message Options and Advanced Features 2337

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - Re-Registration 2339
    - ***Additional Features 2339***
    - Considering Encryption 2342
    - Issues with GCM 2342
    - Amazon Simple Notification Service and GCM 2344
- Basic Use of Sensors
    - Prerequisites 2345
    - The Sensor Abstraction Model 2345
    - Considering Rates 2346
    - Reading Sensors 2347
    - Batching Sensor Readings 2357
- Printing and Document Generation
    - Prerequisites 2360
    - The Android Print System 2360
    - About the Sample App 2361
    - Printing a Bitmap 2362
    - Printing an HTML Document 2364
    - Printing a PDF File 2368
    - Printing Using a Canvas 2376
    - Print Jobs 2378
    - Printing, Threads, and Services 2379
    - Printing Prior to Android 4.4 2381
    - HTML Generation 2382
    - PDF Generation Options 2386
- Other System Settings and Services
    - Prerequisites 2387
    - Setting Expectations 2387
    - Can You Hear Me Now? OK, How About Now? 2392
    - The Rest of the Gang 2395
- Dealing with Different Hardware
    - Prerequisites 2397
    - ***Filtering Out Devices 2397***
    - Runtime Capability Detection 2400
    - Dealing with Device Bugs 2401
- Responding to URLs
    - Prerequisites 2403
    - Manifest Modifications 2403
    - Creating a Custom URL 2405
    - Reacting to the Link 2406
- Plugin Patterns
    - Prerequisites 2409

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - Definitions, Scenarios, and Scope 2409
    - The Keys to Any Plugin System 2410
    - Case Study: DashClock 2418
    - Other Plugin Examples 2421
- PackageManager Tricks
    - Prerequisites 2439
    - Asking Around 2439
    - Preferred Activities 2443
    - Middle Management 2448
- Searching with SearchManager
    - Prerequisites 2451
    - Hunting Season 2451
    - Search Yourself 2453
    - Searching for Meaning In Randomness 2460
    - May I Make a Suggestion? 2461
    - Putting Yourself (Almost) On Par with Google 2465
- Remote Services and the Binding Pattern
    - Prerequisites 2471
    - The Binding Pattern 2472
    - When IPC Attacks! 2478
    - Service From Afar 2480
    - Servicing the Service 2486
    - Thinking About Security 2491
    - The “Everlasting Service” Anti-Pattern 2491
- Advanced Manifest Tips
    - Prerequisites 2493
    - Just Looking For Some Elbow Room 2493
    - Using an Alias 2500
    - Getting Meta (Data) 2502
- Miscellaneous Integration Tips
    - Prerequisites 2507
    - Take the Shortcut 2507
    - Homing Beacons for Intents 2514
- Reusable Components
    - Prerequisites 2515
    - Where Do I Find Them? 2515
    - How Are They Packaged? 2516
    - How Do I Create Them? 2517
    - Other Considerations for Publishing Reusable Code 2520
- The Role of Scripting Languages
    - Prerequisites 2525

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - All Grown Up 2525
    - Following the Script 2526
    - Going Off-Script 2527
- The Scripting Layer for Android
    - Prerequisites 2531
    - The Role of SL4A 2531
    - Getting Started with SL4A 2532
    - Writing SL4A Scripts 2540
    - Running SL4A Scripts 2545
    - Potential Issues 2548
- JVM Scripting Languages
    - Prerequisites 2551
    - Languages on Languages 2551
    - A Brief History of JVM Scripting 2552
    - Limitations 2553
    - SL4A and JVM Languages 2554
    - Embedding JVM Languages 2554
    - Other JVM Scripting Languages 2568
- Advanced Emulator Capabilities
    - Prerequisites 2571
    - x86 Images 2571
    - Hardware Graphics Acceleration 2574
    - Defining New Devices 2577
    - Keyboard Behavior 2580
    - Headless Operation 2580
- Using Lint
    - Prerequisites 2581
    - What It Is 2581
    - When It Runs 2582
    - What to Fix 2585
    - What to Configure 2585
- Using Hierarchy View
    - Prerequisites 2591
    - Launching Hierarchy View 2591
    - Viewing the View Hierarchy 2592
    - ViewServer 2595
- Using DDMS
    - Prerequisites 2597
    - Starting DDMS 2597
    - File Push and Pull 2598
    - Screenshots 2599

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - Location Updates 2599
    - Placing Calls and Messages 2600
- Issues with Speed
    - Prerequisites 2603
    - Getting Things Done 2603
    - Your UI Seems... Janky 2604
    - Not Far Enough in the Background 2604
    - Playing with Speed 2605
- Finding CPU Bottlenecks
    - Prerequisites 2607
    - Traceview 2608
    - Other General CPU Measurement Techniques 2617
    - UI "Jank" Measurement 2619
- Focus On: NDK
    - Prerequisites 2635
    - The Role of the NDK 2636
    - NDK Installation and Project Setup 2639
    - Writing Your Makefile(s) 2643
    - Building Your Library 2644
    - Using Your Library Via JNI 2645
    - Building and Deploying Your Project 2650
    - ***Gradle and the NDK 2652***
- Improving CPU Performance in Java
    - Prerequisites 2659
    - Reduce CPU Utilization 2659
    - Reduce Time on the Main Application Thread 2664
    - Improve Throughput and Responsiveness 2672
- Finding and Eliminating Jank
    - Prerequisites 2675
    - The Case: ThreePaneDemoBC 2675
    - Are We Janky? 2676
    - Finding the Source of the Jank 2676
    - Where Things Went Wrong 2686
    - Removing the Jank 2687
- Issues with Bandwidth
    - Prerequisites 2689
    - You're Using Too Much of the Slow Stuff 2690
    - You're Using Too Much of the Expensive Stuff 2690
    - You're Using Too Much of Somebody Else's Stuff 2691
    - You're Using Too Much... And There Is None 2692
- Focus On: TrafficStats

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - Prerequisites 2693
    - TrafficStats Basics 2693
    - Example: TrafficMonitor 2695
    - Other Ways to Employ TrafficStats 2703
- Measuring Bandwidth Consumption
    - Prerequisites 2705
    - On-Device Measurement 2705
    - Off-Device Measurement 2707
    - Tactical Measurement in DDMS 2709
- Being Smarter About Bandwidth
    - Prerequisites 2713
    - Bandwidth Savings 2713
    - Bandwidth Shaping 2719
    - Avoiding Metered Connections 2722
- Issues with Application Heap
    - Prerequisites 2725
    - You Are in a Heap of Trouble 2726
    - Determining Your Heap Size At Runtime 2726
    - Fragments of Memory 2727
    - Getting a Trim 2728
    - Warning: Contains Graphic Images 2729
    - Releasing SQLite Memory 2740
    - In Too Deep (on the Stack) 2740
- Finding Memory Leaks with MAT
    - Prerequisites 2743
    - Setting Up MAT 2743
    - Getting Heap Dumps 2744
    - Basic MAT Operation 2750
    - Some Leaks and Their MAT Analysis 2757
- Issues with System RAM
    - Prerequisites 2765
    - Can't We All Just Get Along? 2765
    - Contributors to System RAM Consumption 2766
    - Measuring System RAM Consumption: Tools 2767
    - Measuring System RAM Consumption: Runtime 2782
    - Learn To Let Go (Of Your Heap) 2783
- Issues with Battery Life
    - Prerequisites 2785
    - You're Getting Blamed 2786
    - Not All Batteries Are Created Equal 2787
    - Stretching Out the Last mWh 2787

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Power Measurement Options
  - Prerequisites 2789
  - batterystats and the Battery Historian 2790
  - The Qualcomm Tool (That Must Not Be Named) 2800
  - PowerTutor 2801
  - Battery Screen in Settings Application 2805
  - BatteryInfo Dump 2807
- Sources of Power Drain
  - Prerequisites 2811
  - Screen 2812
  - Disk I/O 2813
  - WiFi and Mobile Data 2814
  - GPS 2817
  - Camera 2818
  - Additional Sources 2818
- Addressing Application Size Issues
  - Prerequisites 2821
  - Java Code, and the 64K Method Limit 2821
  - Native Code 2826
  - Images 2828
  - APK Expansion Files 2829
- The Role of Alternative Environments
  - Prerequisites 2831
  - In the Beginning, There Was Java... 2832
  - ... And It Was OK 2832
  - Bucking the Trend 2833
  - Support, Structure 2833
  - Caveat Developer 2834
- HTML5
  - Prerequisites 2835
  - Offline Applications 2835
  - Web Storage 2842
  - Going To Production 2845
  - Issues You May Encounter 2846
  - HTML5: The Baseline 2849
- PhoneGap
  - Prerequisites 2851
  - What Is PhoneGap? 2851
  - Using PhoneGap 2854
  - PhoneGap and the Checklist Sample 2859
  - Issues You May Encounter 2864

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

    - For More Information 2867
- Other Alternative Environments
    - Prerequisites 2869
    - Rhodes 2869
    - Flash, Flex, and AIR 2870
    - JRuby and Ruboto 2870
    - App Inventor 2871
    - Titanium Mobile 2873
    - Other JVM Compiled Languages 2874
- Anti-Patterns
    - Prerequisites 2877
    - Leak Threads... Or Things Attached to Threads 2877
    - Use Large Heap Unnecessarily 2879
    - Misuse the MENU Button 2881
    - Interfere with Navigation 2882
    - Use android:sharedUserId 2884
    - Implement a “Quit” Button 2885
    - Terminate Your Process 2887
    - Try to Hide from the User 2888
    - Use Multiple Processes 2889
    - Hog System Resources 2891
- Widget Catalog: AdapterViewFlipper
    - Key Usage Tips 2893
    - A Sample Usage 2894
    - Visual Representation 2894
- Widget Catalog: CalendarView
    - Key Usage Tips 2895
    - A Sample Usage 2896
    - Visual Representation 2897
- Widget Catalog: DatePicker
    - Key Usage Tips 2899
    - A Sample Usage 2900
    - Visual Representation 2901
- Widget Catalog: ExpandableListView
    - Key Usage Tips 2905
    - A Sample Usage 2906
    - Visual Representation 2912
- Widget Catalog: SeekBar
    - Key Usage Tips 2915
    - A Sample Usage 2915
    - Visual Representation 2917

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Widget Catalog: SlidingDrawer
  - Key Usage Tips 2919
  - A Sample Usage 2920
  - Visual Representation 2921
- Widget Catalog: StackView
  - Key Usage Tips 2923
  - A Sample Usage 2924
  - Visual Representation 2925
- Widget Catalog: TabHost and TabWidget
  - Deprecation Notes 2927
  - Key Usage Tips 2927
  - A Sample Usage 2928
  - Visual Representation 2930
- Widget Catalog: TimePicker
  - Key Usage Tips 2933
  - A Sample Usage 2933
  - Visual Representation 2935
- Widget Catalog: ViewFlipper
  - Key Usage Tips 2937
  - A Sample Usage 2938
  - Visual Representation 2939
- Device Catalog: Kindle Fire
  - Prerequisites 2941
  - Introducing the Kindle Fire series 2941
  - What Features and Configurations Does It Use? 2942
  - What Is Really Different? 2944
  - Getting Your Development Environment Established 2950
  - How Does Distribution Work? 2952
  - Amazon Equivalents of Google Services 2953
  - Getting Help with the Kindle Fire 2954
- Device Catalog: BlackBerry
  - I Thought BlackBerry Had Their Own OS? 2955
  - What Else Is Different? 2956
  - What Are We Making? 2959
  - Getting Your Development Environment Established 2959
  - How Does Distribution Work? 2961
- Device Catalog: Wrist Wearables
  - Prerequisites 2964
  - Divvying Up the Wearables Space 2964
  - Example Wrist Wearables 2965
  - Strategic Considerations 2968

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Tactical Considerations 2970
- What About Android Wear? 2973

- Device Catalog: Android TV
  - ***Prerequisites 2975***
  - ***Hey, Wait a Minute... I Thought the Name Was "Google TV"? 2975***
  - ***Some Android TV Hardware 2976***
  - ***What Features and Configurations Does It Use? 2978***
  - ***What Is Really Different? 2979***
  - ***Getting Your Development Environment Established 2981***
  - ***How Does Distribution Work? 2983***
- Device Catalog: Amazon Fire TV and Fire TV Stick
  - ***Prerequisites 2987***
  - ***Introducing the Fire TV Devices 2987***
  - ***What Features and Configurations Do They Use? 2993***
  - ***What Is Really Different? 2995***
  - ***Casting and Fire TV 2996***
  - ***Getting Your Development Environment Established 2997***
  - ***Working with the Remote and Controller 2999***
  - ***How Does Distribution Work? 3001***
  - Getting Help 3001
- CWAC Libraries
  - cwac-adapter 3003
  - cwac-camera 3003
  - cwac-colormixer 3004
  - cwac-layouts 3004
  - ***cwac-mediarouter 3004***
  - cwac-merge 3004
  - cwac-pager 3005
  - cwac-presentation 3005
  - cwac-provider 3005
  - cwac-richedit 3005
  - cwac-sacklist 3006
  - cwac-security 3006
  - cwac-strictmodeex 3006
  - cwac-wakeful 3006

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Preface

Welcome to the Book!

Thanks!

Thanks for your interest in developing applications for Android! Android has grown from nothing to arguably the world's most popular smartphone OS in a few short years. Whether you are developing applications for the public, for your business or organization, or are just experimenting on your own, I think you will find Android to be an exciting and challenging area for exploration.

And, most of all, thanks for your interest in this book! I sincerely hope you find it useful and at least occasionally entertaining.

The Book's Structure

As you may have noticed, this is a rather large book.

To make the equivalent of 2,800+ pages of material manageable, the chapters are divided into the *core* chapters and a series of *trails*.

The core chapters represent many key concepts that Android developers need to understand in order to build an app. While an occasional "nice to have" topic will drift into the core — to help illustrate a point, for example — the core chapters generally are fairly essential.

The core chapters are designed to be read in sequence and will interleave both traditional technical book prose with tutorial chapters, to give you hands-on experience with the concepts being discussed. Most of the tutorials can be skipped,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

though the first two — covering setting up your SDK environment and creating a project – everybody should read.

The bulk of the chapters are divided into trails, covering some particular general topic, from data storage to advanced UI effects to performance measurement and tuning. Each trail will have several chapters. However, those chapters, and the trails themselves, are not necessarily designed to be read in any order. Each chapter in the trails will point out prerequisite chapters or concepts that you will want to have covered in advance. Hence, these chapters are mostly reference material, for when you specifically want to learn something about a specific topic.

The core chapters will link to chapters in the trails, to show you where you can find material related to the chapter you just read. So between the book's table of contents, this preface, the search tool in your digital book reader, and the cross-chapter links, you should have plenty of ways of finding the material you want to read.

You are welcome to read the entire book front-to-back if you wish. The trails will appear after the core chapters. Those trails will be in a reasonably logical order, though you may have to hop around a bit to cover all of the prerequisites.

The Trails

Here is a list of all of the trails and the chapters that pertain to those trails, in order of appearance (except for those appearing in the list multiple times, where they span major categories):

Code Organization and Gradle

- Working with Library Projects
- Gradle and Legacy Projects
- Gradle and Tasks
- Gradle and the New Project Structure
- Gradle and Dependencies
- Manifest Merger Rules
- Signing Your App
- Distribution
- Advanced Gradle for Android Tips

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Testing

- JUnit and Android
- Testing with JUnit4
- MonkeyRunner and the Test Monkey
- Testing with UIAutomator

Advanced UI

- Introducing GridLayout
- Dialogs and DialogFragments
- Advanced ListViews
- Action Bar Navigation
- Action Modes and Context Menus
- Other Advanced Action Bar Techniques
- AppCompat: The Official Action Bar Backport
- ActionBarSherlock
- Implementing a Navigation Drawer
- Advanced Uses of WebView
- The Input Method Framework
- Fonts
- Rich Text
- Animators
- Legacy Animations
- Custom Drawables
- Mapping with Maps V2
- Crafting Your Own Views
- Custom Dialogs and Preferences
- Progress Indicators
- Advanced Notifications
- More Fun with Pagers
- Focus Management and Accessibility
- Miscellaneous UI Tricks
- Event Bus Alternatives

Home Screen Effects

- Home Screen App Widgets
- Adapter-Based App Widgets

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Data Storage and Retrieval

- Content Provider Theory
- Content Provider Implementation Patterns
- The Loader Framework
- The ContactsContract Provider
- The CalendarContract Provider
- The MediaStore Provider
- Consuming Documents
- Providing Documents
- Encrypted Storage
- Packaging and Distributing Data
- Advanced Database Techniques
- Miscellaneous Network Capabilities

Media

- Audio Playback
- Audio Recording
- Video Playback
- Using the Camera via 3rd-Party Apps
- Working Directly with the Camera
- The MediaStore Provider
- Media Routes
- Supporting External Displays
- Google Cast and ChromeCast
- The "10 Foot UI"
- Putting the TVs All Together: Decktastic
- Creating a MediaRouteProvider

Security

- SSL
- Encrypted Storage
- Advanced Permissions
- Restricted Profiles and UserManager
- Tapjacking
- Miscellaneous Security Techniques

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Hardware and System Services

- Accessing Location-Based Services
- The Fused Location Provider
- Working with the Clipboard
- Telephony
- Working With SMS
- NFC
- Device Administration
- PowerManager and WakeLocks
- JobScheduler
- Push Notifications with GCM
- Basic Use of Sensors
- Printing and Document Generation
- Other System Settings and Services
- Dealing with Different Hardware

Integration and Introspection

- Responding to URLs
- Plugin Patterns
- PackageManager Tricks
- Searching with SearchManager
- Remote Services and the Binding Pattern
- Advanced Manifest Tips
- Miscellaneous Integration Tips
- Reusable Components

Scripting Languages

- The Role of Scripting Languages
- The Scripting Layer for Android
- JVM Scripting Languages

Other Tools

- Advanced Emulator Capabilities
- Using Lint
- Using Hierarchy View
- Using DDMS

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Finding CPU Bottlenecks
- Finding Memory Leaks with MAT

Tuning Android Applications

- Issues with Speed
- Finding CPU Bottlenecks
- NDK
- Improving CPU Performance in Java
- Finding and Eliminating Jank
- Issues with Bandwidth
- Focus On: TrafficStats
- Measuring Bandwidth Consumption
- Being Smarter About Bandwidth
- Issues with Application Heap
- Finding Memory Leaks with MAT
- Issues with System RAM
- Issues with Battery Life
- Other Power Measurement Options
- Sources of Power Drain
- Addressing Application Size Issues

Alternatives for App Development

- The Role of Alternative Environments
- HTML5
- PhoneGap
- Other Alternative Environments

Miscellaneous Topics

- Anti-Patterns

Widget Catalog

- AdapterViewFlipper
- CalendarView
- DatePicker
- ExpandableListView
- SeekBar

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- SlidingDrawer
- StackView
- TabHost
- TimePicker
- ViewFlipper

Device Catalog

- Kindle Fire
- BlackBerry
- Wrist Wearables
- Google TV
- Amazon Fire TV

Appendices

- Appendix A: CWAC Libraries

About the Updates

This book is updated frequently, typically every 6–8 weeks.

Each release has notations to show what is new or changed compared with the immediately preceding release:

- The Table of Contents shows sections with changes in bold-italic font
- Those sections have changebars on the right to denote specific paragraphs that are new or modified

Warescription

You (hopefully) are reading this digital book by means of a Warescription.

The Warescription entitles you, for the duration of your subscription, to digital editions of this book and its updates, in PDF, EPUB, and Kindle (MOBI/KF8) formats. You also have access to a version of the book as its own Android APK file, complete with high-speed full-text searching. You also have access to other titles that CommonsWare may publish during that subscription period.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Each subscriber gets personalized editions of all editions of each title. That way, your books are never out of date for long, and you can take advantage of new material as it is made available. For example, when new releases of the Android SDK are made available, this book will be quickly updated to be accurate with changes in the APIs.

However, you can only download the books if either you have an active Warescription, or until the book is updated after your Warescription expires. Hence, **please download your updates as they come out**. You can find out when new releases of this book are available via:

1. The commonsguy Twitter feed
2. The CommonsBlog
3. The Warescription newsletter, which you can subscribe to off of your Warescription page
4. Just check back on the Warescription site every month or two

Subscribers also have access to other benefits, including:

- "Office hours" — online chats to help you get answers to your Android application development questions. You will find a calendar for these on your Warescription page.
- A Stack Overflow "bump" service, to get additional attention for a question that you have posted there that does not have an adequate answer.
- 80% off of live webinars hosted by Mark Murphy, the author of this book, on Android application development topics.

Book Bug Bounty

Find a problem in one of our books? Let us know!

Be the first to report a unique concrete problem in the current digital edition, and we will extend your Warescription by six months as a bounty for helping us deliver a better product.

By "concrete" problem, we mean things like:

1. Typographical errors
2. Sample applications that do not work as advertised, in the environment described in the book

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

3. Factual errors that cannot be open to interpretation

By "unique", we mean ones not yet reported. Be sure to check the book's errata page, though, to see if your issue has already been reported. One coupon is given per email containing valid bug reports.

We appreciate hearing about "softer" issues as well, such as:

1. Places where you think we are in error, but where we feel our interpretation is reasonable
2. Places where you think we could add sample applications, or expand upon the existing material
3. Samples that do not work due to "shifting sands" of the underlying environment (e.g., changed APIs with new releases of an SDK)

However, those "softer" issues do not qualify for the formal bounty program.

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to bounty@commonsware.com.

Source Code and Its License

The source code samples shown in this book are available for download from the book's GitHub repository. All of the Android projects are licensed under the Apache 2.0 License, in case you have the desire to reuse any of it.

If you wish to use the source code from the GitHub repository, please follow the instructions on that repository's home page for details of how to use the projects in various development environments, notably Eclipse and Android Studio.

If you are using Eclipse, please do **NOT** import all of the projects from the repo into your main workspace. There are *hundreds* of these projects, and they may cause your Eclipse environment to become very slow, particularly when starting it up. Instead, import only those specific projects that you want to work with "live" as opposed to simply reading about them in the book.

Copying source code directly from the book, in the PDF editions, works best with Adobe Reader, though it may also work with other PDF viewers. Some PDF viewers, for reasons that remain unclear, foul up copying the source code to the clipboard when it is selected.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on *1 January 2019*. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license, visit the Creative Commons Web site

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Acknowledgments

I would like to thank the Android team, not only for putting out a good product, but for invaluable assistance on the Android Google Groups and Stack Overflow.

I would also like to thank the thousands of readers of past editions of this book, for their feedback, bug reports, and overall support.

Of course, thanks are also out to the overall Android ecosystem, particularly those developers contributing their skills to publish libraries, write blog posts, answer support questions, and otherwise contribute to the strength of Android.

Portions of this book are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Core Chapters

Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Key Android Concepts

No doubt, you are in a hurry to get started with Android application development. After all, you are reading this book, aimed at busy coders.

However, before we dive into getting tools set up and starting in on actual programming, it is important that we "get on the same page" with respect to several high-level Android concepts. This will simplify further discussions later in the book.

Android Applications

This book is focused on writing Android applications. An application is something that a user might install from the Play Store or otherwise download to their device. That application should have some user interface, and it might have other code designed to work in the background (multi-tasking).

This book is not focused on modifications to the Android firmware, such as writing device drivers. For that, you will need to seek other resources.

This book assumes that you have some hands-on experience with Android devices, and therefore you are familiar with buttons like HOME and BACK, the built-in Settings application, the concept of a home screen and launcher, and so forth. If you have never used an Android device, you are strongly encouraged to get one (e.g., a used one on eBay, Craigslist, etc.) and spend some time with it before starting in on learning Android application development.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Programming Language

The vast majority of Android applications are written exclusively in Java. Hence, that is what this book will spend most of its time on and will demonstrate with a seemingly infinite number of examples.

However, there are other options:

- You can write parts of the app in C/C++, for performance gains, porting over existing code bases, etc.
- You can write an entire app in C/C++, mostly for games using OpenGL for 3D animations
- You can write the guts of an app in HTML, CSS, and JavaScript, using tools to package that material into an Android application that can be distributed through the Play Store and similar venues
- And so on

Coverage of these non-Java alternatives will be found in the trails of this book, as the bulk of this book is focused on Java.

The author assumes that you know Java at this point. If you do not, you will need to learn Java before you go much further. You do not need to know *everything* about Java, as Java is vast. Rather, focus on:

- Language fundamentals (flow control, etc.)
- Classes and objects
- Methods and data members
- Public, private, and protected
- Static and instance scope
- Exceptions
- Threads
- Collections
- Generics
- File I/O
- Reflection
- Interfaces

The links are to Wikibooks material on those topics, though there are countless other Java resources for you to consider.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Components

When you first learned Java — whether that was yesterday or back when dinosaurs roamed the Earth — you probably started off with something like this:

```
class SillyApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

In other words, the entry point into your application was a `public static void` method named `main()` that took a `String` array of arguments. From there, you were responsible for doing whatever was necessary.

However, there are other patterns used elsewhere in Java. For example, you do not usually write a `main()` method when writing a Java servlet. Instead, you extend a particular class supplied by a framework (e.g., `HttpServlet`) to create a component, then write some metadata that enumerates your components and tell the framework when and how to use them (e.g., `WEB.XML`).

Android apps are closer in spirit to the servlet approach. You will not write a `public static void main()` method. Instead, you will create subclasses of some Android-supplied base classes that define various application components. In addition, you will create some metadata that tells Android about those subclasses.

There are four types of components, all of which will be covered extensively in this book:

Activities

The building block of the user interface is the *activity*. You can think of an activity as being the Android analogue for the window or dialog in a desktop application, or the page in a classic Web app. It represents a chunk of your user interface and, in some cases, a discrete entry point into your app (i.e., a way for other apps to link to your app).

Normally, an activity will take up most of the screen, leaving space for some "chrome" bits like the clock, signal strength indicators, and so forth.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

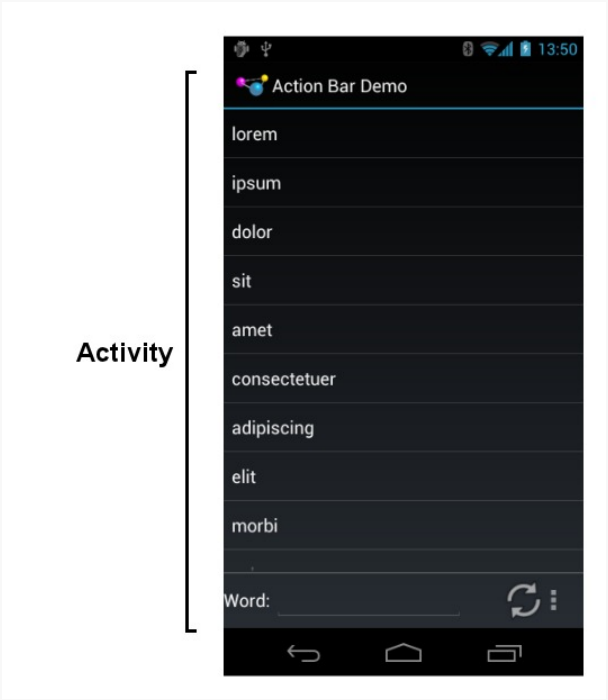


Figure 1: Activity on the screen

Services

Activities are short-lived and can be shut down at any time, such as when the user presses the BACK button. *Services*, on the other hand, are designed to keep running, if needed, independent of any activity, for a moderate period of time. You might use a service for checking for updates to an RSS feed, or to play back music even if the controlling activity is no longer operating. You will also use services for scheduled tasks (akin to Linux or OS X "cron jobs") and for exposing custom APIs to other applications on the device, though the latter is a relatively advanced capability.

Content Providers

Content providers provide a level of abstraction for any data stored on the device that is accessible by multiple applications. The Android development model encourages you to make your own data available to other applications, as well as your own — building a content provider lets you do that, while maintaining a degree of control over how your data gets accessed.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Broadcast Receivers

The system, or applications, will send out *broadcasts* from time to time, for everything from the battery getting low, to when the screen turns off, to when connectivity changes from WiFi to mobile data. A broadcast receiver can arrange to listen for these broadcasts and respond accordingly.

Widgets, Containers, Resources, and Fragments

Most of the focus on Android application development is on the UI layer and activities. Most Android activities use what is known as "the widget framework" for rendering their user interface, though you are welcome to use the 2D (`Canvas`) and 3D (OpenGL) APIs as well for more specialized GUIs.

In Android terms, a *widget* is the "micro" unit of user interface. Fields, buttons, labels, lists, and so on are all widgets. Your activity's UI, therefore, is made up of one or more of these widgets. For example, here we see label (`TextView`), field (`EditText`), and push-button (`Button`) widgets:

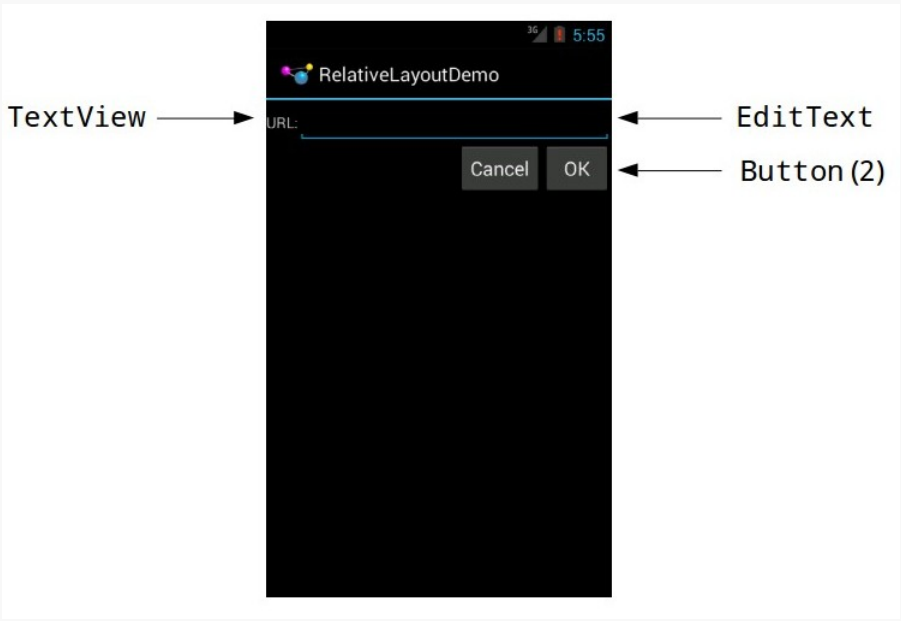


Figure 2: Activity with widgets

If you have more than one widget — which is fairly typical — you will need to tell Android how those widgets are organized on the screen. To do that, you will use

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

various container classes referred to as *layout managers*. These will let you put things in rows, columns, or more complex arrangements as needed.

To describe how the containers and widgets are connected, you will typically create a *layout resource file*. *Resources* in Android refer to things like images, strings, and other material that your application uses but is not in the form of some programming language source code. UI layouts are another type of resource. You will create these layouts either using a structured tool, such as an IDE's drag-and-drop GUI builder, or by hand in XML form.

Sometimes, your UI will work across all sorts of devices: phones, tablets, televisions, etc. Sometimes, your UI will need to be tailored for different environments. You will be able to put resources into *resource sets* that indicate under what circumstances those resources can be used (e.g., use these for normal-sized screens, but use those for larger screens).

Sometimes, supporting larger screens means you will want to "snap together" parts of your smaller-screen UI. For example, Gmail on a tablet will show your list of labels, the list of conversations in a selected label, and the list of messages in a selected conversation, all in one activity. However, Gmail on a phone cannot do that, as there is not enough screen space, so it shows each of those (labels, conversations, messages) in separate activities. Android supplies a construct called the *fragment* to help make it easier for you to implement these sorts of effects.

We will be examining all of these concepts, in much greater detail, as we get deeper into the book.

Apps and Packages

Given a bucket of source code and a basket of resources, the Android build tools will give you an application as a result. The application comes in the form of an *APK file*. It is that APK file that you will upload to the Play Store or distribute by other means.

Each Android application has a package name. A package name must fulfill three requirements:

1. It must be a valid Java package name, as some Java source code will be generated by the Android build tools in this package
2. No two applications can exist on a device at the same time with the same package

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

3. No two applications can be uploaded to the Play Store having the same package

Note that sometimes you will see reference to an "application ID" — this is the role of the package name from the second and third items in the list.

When you create your Android project — the repository of that source code and those resources — you will declare what package name is to be used for your app. Typically, you will pick a package name following the Java package name "reverse domain name" convention (e.g., `com.commonsware.android.foo`). That way, the domain name system ensures that your package name prefix (`com.commonsware`) is unique, and it is up to you to ensure that the rest of the package name distinguishes one of your apps from any other.

Android Devices

There are well in excess of one billion Android devices in use today, representing thousands of different models from dozens of different manufacturers. Android itself has evolved since Android 1.0 in 2008. Between different device types and different Android versions, many a media pundit has lobbed the term "fragmentation" at Android, suggesting that creating apps that run on all these different environments is impossible.

In reality, it is not that bad. Some apps will have substantial trouble, but most apps will work just fine if you follow the guidance presented in this book and in other resources.

Types

Android devices come in all shapes, sizes, and colors. However, there are four dominant "form factors":

- the phone
- the tablet
- the television (TV)
- the wearable (smart watches, Google Glass, etc.)

You will often hear developers and pundits refer to these form factors, and this book will do so from time to time as well. However, it is important that you understand that Android has no built-in concept of a device being a "phone" or a "tablet" or a

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

"TV". Rather, Android distinguishes devices based on capabilities and features. So, you will not see an `isPhone()` method anywhere, though you can ask Android:

- what is the screen size?
- does the device have telephony capability?
- etc.

Similarly, as you build your applications, rather than thinking of those four form factors, focus on what capabilities and features you need. Not only will this help you line up better with how Android wants you to build your apps, but it will make it easier for you to adapt to other form factors that will come about such as:

- airplane seat-back entertainment centers
- in-car navigation and entertainment devices
- and so on

The Emulator

While there are over a billion Android devices representing thousands of models, probably you do not have one of each model. You may only have a single piece of Android hardware. And if you do not even have that, you most certainly will want to acquire one before trying to publish an Android app.

To help fill in the gaps between the devices you have and the devices that are possible, the Android developer tools ship an *emulator*. The emulator behaves like a piece of Android hardware, but it is a program you run on your development machine. You can use this emulator to emulate many different devices, with different screen sizes and Android OS versions, by creating one or more Android virtual devices, or *AVDs*.

In an upcoming chapter, we will discuss how you install the Android developer tools and how you will be able to create these AVDs and run the emulator.

OS Versions and API Levels

Android has come a long way since the early beta releases from late 2007. Each new Android OS version adds more capabilities to the platform and more things that developers can do to exploit those capabilities.

Moreover, the core Android development team tries very hard to ensure forwards and backwards compatibility. An app you write today should work unchanged on

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

future versions of Android (forwards compatibility), albeit perhaps missing some features or working in some sort of "compatibility mode". And there are well-trod paths for how to create apps that will work both on the latest and on previous versions of Android (backwards compatibility).

To help us keep track of all the different OS versions that matter to us as developers, Android has *API levels*. A new API level is defined when an Android version ships that contains changes that affect developers. When you create an emulator AVD to test your app, you will indicate what API level that emulator should emulate. When you distribute your app, you will indicate the oldest API level your app supports, so the app is not installed on older devices.

At the time of this writing, the API levels of significance to most Android developers are:

- API Level 10 (Android 2.3.3)
- API Level 15 (Android 4.0.3)
- API Level 16 (Android 4.1)
- API Level 17 (Android 4.2)
- API Level 18 (Android 4.3)
- API Level 19 (Android 4.4)

Here, "of significance" refers to API levels that have a reasonable number of Android devices — 5% or more, as reported by the "Platform Versions" dashboard chart.

The latest API level for most form factors is 21, representing Android 5.0. There is an API Level 20, which is the version of Android running on the first-generation Android Wear devices. Unless you are specifically developing apps for Wear, you will not be worrying much about API Level 20.

Dalvik

In terms of Android, Dalvik is a virtual machine (VM). Virtual machines are used by many programming languages, such as Java, Perl, and Smalltalk. The Dalvik VM is designed to work much like a Java VM, but optimized for embedded Linux environments.

So, what really goes on when somebody writes an Android application is:

1. Developers write Java-syntax source code, leveraging class libraries published by the Android project and third parties.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. Developers compile the source code into Java VM bytecode, using the `javac` compiler that comes with the Java SDK.
3. Developers translate the Java VM bytecode into Dalvik VM bytecode, which is packaged with other files into a ZIP archive with the `.apk` extension (the APK file).
4. An Android device or emulator runs the APK file, causing the bytecode to be executed by an instance of a Dalvik VM.

From your standpoint, most of this is hidden by the build tools. You pour Java source code into the top, and the APK file comes out the bottom.

However, there will be places from time to time where the differences between the Dalvik VM and the traditional Java VM will affect application developers, and this book will point out some of them where relevant.

Note that Android is moving to a new runtime environment, called ART. However, the "Dalvik" term will still be used for the bytecode that is generated as part of building an APK.

Processes and Threads

When your application runs, it will do so in its own process. This is not significantly different than any other traditional operating system. Part of Dalvik's magic is making it possible for many processes to be running many Android applications at one time without consuming ridiculous amounts of RAM.

Android will also set up a batch of threads for running your app. The thread that your code will be executed upon, most of the time, is variously called the "main application thread" or the "UI thread". You do not have to set it up, but, as we will see later in the book, you will need to pay attention to what you do and do not do on that thread. You are welcome to fork your own threads to do work, and that is fairly common, though in some places Android handles that for you behind the scenes.

Don't Be Scared

Yes, this chapter threw a lot of terms at you. We will be going into greater detail on all of them in this book. However, Android is like a jigsaw puzzle with lots of interlocking pieces. To be able to describe one concept in detail, we will need to at least reference some of the others. Hence, this chapter was meant to expose you to terms, in hopes that they will sound vaguely familiar as we dive into the details.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Choosing Your Development Toolchain

Before you go much further in your Android endeavors (or, possibly, endeavours, depending upon your preferred spelling), you will need to determine what toolchain you will use to build your Android applications.

Android Studio

The next-generation Google-backed Android IDE is Android Studio. Based off of IntelliJ IDEA, Android Studio is the new foundation of Google's efforts to give Android developers top-notch development tools. While it only reached a version 1.0 status in December 2014, Android Studio had been in use for ~18 months prior to that in various early-access and beta stages. While it still has bugs, it is certainly stable enough for app development.

The next chapter contains a section with instructions on how to set up Android Studio.

Eclipse

Eclipse is also a popular IDE, particularly for Java development. Eclipse was Google's original IDE for Android development, by means of the Android Developer Tools (ADT) add-in, which gives the core of Eclipse awareness of Android. The ADT add-in, in essence, takes regular Eclipse operations and extends them to work with Android projects.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note, though, that Google has discontinued maintenance of ADT. The Eclipse Foundation is setting up the "Andmore" project to try to continue work on allowing Eclipse to build Android apps. This book does not cover the Andmore project at this time, and developers are **strongly** encouraged to not use the ADT-enabled Eclipse from Google.

For historical reasons, the next chapter contains a section with instructions on how to set up Eclipse for Android development, as part of getting an overall Android development environment established. Similarly, there will be Eclipse instructions in the tutorials and notes about using Eclipse elsewhere in the book. These will be removed from a future edition of the book, or perhaps migrated over to Andmore.

IntelliJ IDEA

While Android Studio is based off of IntelliJ IDEA, you can still use the original IntelliJ IDEA for Android app development. A large subset of the Android Studio capabilities are available in the Android plugin for IDEA. Plus, the commercial IDEA Ultimate Edition will go beyond Android Studio in many areas outside of Android development.

Command-Line Builds via Gradle for Android

And, of course, you do not need to use an IDE at all. While this may sound sacrilegious to some, IDEs are not the only way to build applications. Much of what is accomplished via the ADT can be accomplished through command-line equivalents, meaning a shell and an editor is all you truly need. For example, the author of this book did not use an IDE for Android development until 2011.

The recommended way to build Android apps outside of an IDE is by means of Gradle. Google has published a Gradle plugin that teaches Gradle how to build Android apps. Android Studio itself uses Gradle for its builds, so a single build configuration (e.g., `build.gradle` files) can be used both from an IDE and from a build automation tool like a continuous integration server.

An upcoming chapter gets into more about what Gradle (and the Gradle for Android plugin) are all about.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Yet Other Alternatives

Other IDEs have their equivalents of the ADT, albeit with minimal assistance from Google. For example, NetBeans has support via the NBAndroid add-on, and reportedly this has advanced substantially in the past few years.

You will also hear reference to using Apache Ant for doing command-line builds of Android apps. This has largely been supplanted by Gradle for Android at this time, and support for Apache Ant will end soon. Newcomers to Android are encouraged to not invest time in new work with Apache Ant for Android development projects.

IDEs… And This Book

You are welcome to use Android Studio or Eclipse as you work through this book. You are welcome to use another IDE if you wish. You are even welcome to skip the IDE outright and just use an editor.

This book is focused primarily on demonstrating Android capabilities and the APIs for exploiting those capabilities. Hence, the sample code will work with any IDE. However, this book will cover some Android Studio- and Eclipse-specific instructions, since they are the predominant answers today.

The tutorials will have instructions for both Android Studio and Eclipse.

What We Are Not Covering

In the beginning (a.k.a., 2007), we were lucky to have any means of creating an Android app.

Nowadays, there seems to be no end to the means by which we can create an Android app.

There are a few of these “means”, though, that are specifically out of scope for this book.

App Inventor

You may also have heard of a tool named App Inventor and wonder where it fits in with all of this.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

App Inventor was originally created by an education group within Google, as a means of teaching students how to think about programming constructs (branches, loops, etc.) and create interesting output (Android apps) without classic programming in Java or other syntax-based languages. App Inventor is purely drag-and-drop, both of widgets *and application logic*, the latter by means of "blocks" that snap together to form logic chains.

App Inventor was donated by Google to MIT, which has recently re-opened it to the public.

However, App Inventor is a closed system — at the present time, it does not somehow generate Java code that you can later augment. That limits you to whatever App Inventor is natively capable of doing, which, while impressive in its own right, offers a small portion of the total Android SDK capabilities.

App Generators

There are a seemingly infinite number of "app generators" available as online services. These are designed mostly for creating apps for specific vertical markets, such as apps for restaurants or apps for grocers. The resulting apps are mostly "brochure-ware", with few capabilities beyond a mobile Web site, yet still requiring the user to find, download, and install the app. Few of these generators provide the source code to the generated app, to allow the apps to be customized beyond what the generator generates.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #1 - Installing the Tools

Now, let us get you set up with the pieces and parts necessary to build an Android app.

NOTE: The instructions presented here are accurate as of the time of this writing. However, the tools change rapidly, and so these instructions may be out of date by the time you read this. Please refer to the Android Developers Web site for current instructions, using this as a base guideline of what to expect.

Step #1 - Checking Your Hardware Requirements

Compiling and building an Android application, on its own, is not especially hardware-intensive, except for very large projects. However, there are two commonly-used tools that demand more from your development machine: your IDE and the Android emulator. Of the two, the emulator poses the bigger problem.

The more RAM you have, the better. 8GB or higher is a very good idea if you intend to use an IDE and the emulator together.

A faster CPU is also a good idea. However, the Android emulator only utilizes a single core from your development machine. Hence, it is the single-core speed that matters. The best CPU to use is one that can leverage multiple cores to give what amounts to a faster single core, such as Intel's Core i7 with Turbo Boost. For an emulator simulating a larger-screened device (e.g., tablet, television), a Core i7 that can "boost" up to 3.4GHz makes development much more pleasant. Conversely, a CPU like a Core 2 Duo with a 2.5GHz clock speed results in a tablet emulator that is nearly unusable.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Step #2 - Setting Up Java and 32-Bit Linux Support

When you write Android applications, you typically write them in Java source code. That Java source code is then turned into the stuff that Android actually runs (Dalvik bytecode in an APK file).

You need to obtain and install the official Sun/Oracle Java SE SDK (JDK). You can obtain this from the Oracle Java Web site for Windows, OS X, and Linux. The plain JDK (sans any "bundles") should suffice. Follow the instructions supplied by Oracle or Apple for installing it on your machine. At the time of this writing, Android supports Java 6 and Java 7, though Java 7 is required for certain scenarios and therefore is recommended. Java 8 works, though you may have to do additional work to configure your IDE to have Java 8 emit Java 7-compatible bytecode.

Android also supports the OpenJDK, particularly on Linux environments.

What Android does *not* support are any other Java compilers, including the GNU Compiler for Java (GCJ).

If your development OS is Linux, make sure that you can run 32-bit Linux binaries. This may or may not already be enabled in your Linux distro. For example, on Ubuntu 14.10, you may need to run the following to get the 32-bit binary support installed that is needed by the Android build tools:

```
sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0 lib32stdc++6
```

Step #3 - Install the Developer Tools

As noted in the previous chapter, there are a few developer tools that you can choose from.

There are three major options that this book covers:

1. Use Android Studio
2. Use Eclipse, as a fresh install via the ADT Bundle
3. Use Eclipse, adding in the ADT plugin to an existing Eclipse installation

A fourth option would be to skip either IDE and just use the Android SDK and Gradle for Android directly. This would only be appropriate for fairly expert developers, and so the tutorials do not walk you through this path. This book has

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

several chapters on Gradle, but those chapters are presently designed for developers already experienced in working with Android apps. For the purposes of learning Android, you are better served working with one of the supported IDEs initially. Later on, if you wish to drop the IDE and use Gradle for Android directly, you are welcome to do so.

Options #2 and #3 are close to being mutually exclusive. And, as noted earlier in the book, Eclipse support has been officially dropped by Google and, therefore, you **really** should consider using Android Studio over Eclipse today. Otherwise, you are welcome to choose one or more of these options. The installation instructions for each are described in the sections that follow.

Option #1 - Android Studio

Visit the Android Studio download page, download the ZIP file for your platform, and unZIP it to some likely spot on your hard drive. Windows users who choose to download the self-installing EXE can just run that file.

Android Studio can then be run from the `studio` batch file or shell script from your Android Studio installation's `bin/` directory.

Option #2 - Android ADT Bundle

The Android ADT Bundle contains:

- The Android SDK, which gives you all the tools you need to create and test Android applications. It comes in two parts: the base tools, plus version-specific SDKs and related add-ons.
- A copy of Eclipse, with the Android Developer Tools (ADT) plug-in pre-installed.

From the Android Developer site's SDK area, choose the ADT Bundle ZIP file that is appropriate for your development machine and unZIP it in some likely location.

Option #3 - Install the ADT for Eclipse

To use Eclipse, you will need to install the base Android SDK, then configure Eclipse.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Install the Android SDK

The Android developer tools can be found on the Android Developers Web site.

You will want to click on "Using an Existing IDE" (even if you have not yet installed Eclipse) and download the ZIP or TGZ file presented to you, unpacking it in some likely spot — there is no specific path that is required. Windows users also have the option of running a self-installing EXE file.

Configure Eclipse

If you have not yet installed Eclipse, you will need to do that first. Eclipse can be downloaded from the Eclipse Web site. The "Eclipse IDE for Java Developers" package will work fine. Note that the Android tools require Eclipse 3.7 (Indigo) or newer at the time of this writing.

If you already had Eclipse installed, it is a good idea for you to go in and check your compiler compliance level (Preferences > Java > Compiler). That should be set to 1.6. Notably, this allows the use of `@Override` annotations to indicate methods that are implementing a Java interface, rather than truly overriding a superclass method. This annotation is very common in Java code in Android projects (including many of the samples in this book).

Next, you need to install the Android Developer Tools (ADT) plug-in. To do this, go to Help | Install New Software... in the Eclipse main menu. Then, click the Add button to add a new source of plug-ins. Give it some name (e.g., Android) and supply the following URL: `https://dl-ssl.google.com/android/eclipse/`. That should trigger Eclipse to download the roster of plug-ins available from that site:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

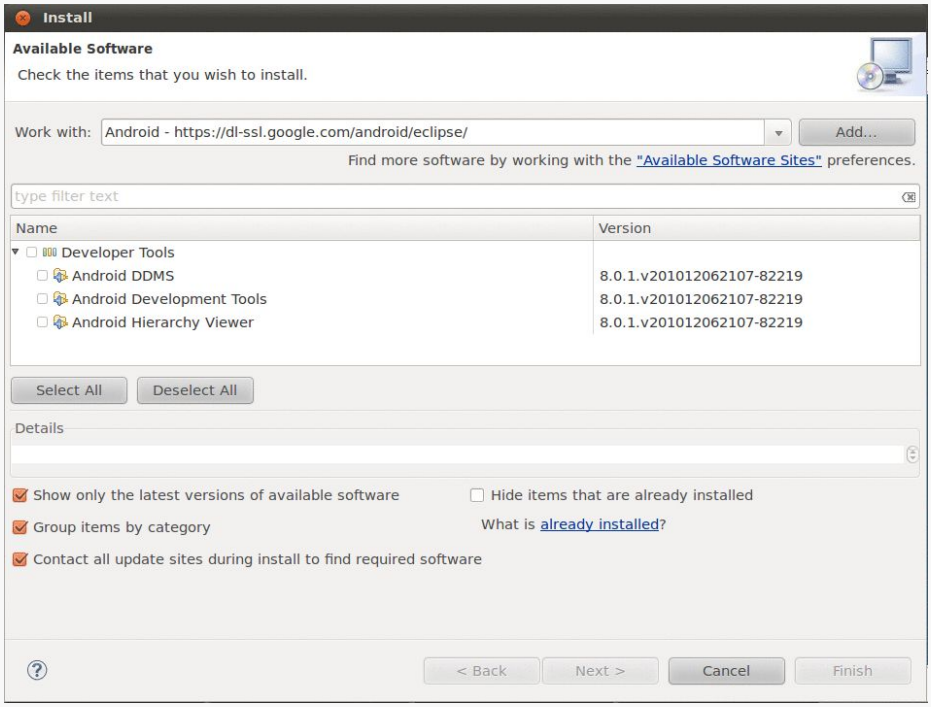


Figure 3: Eclipse ADT plug-in installation

Check the checkbox to the left of "Developer Tools" and click the Next button. Follow the rest of the wizard to review the tools to be downloaded and their respective license agreements. When the Finish button is enabled, click it, and Eclipse will download and install the plug-ins. When done, Eclipse will ask to restart — please let it.

Then, you need to teach ADT where your Android SDK installation is from the preceding section. This should occur on your next restart of Eclipse, via a "welcome wizard". Otherwise, to do this, choose Window | Preferences from the Eclipse main menu (or the equivalent Preferences option for OS X). Click on the Android entry in the list on the left:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

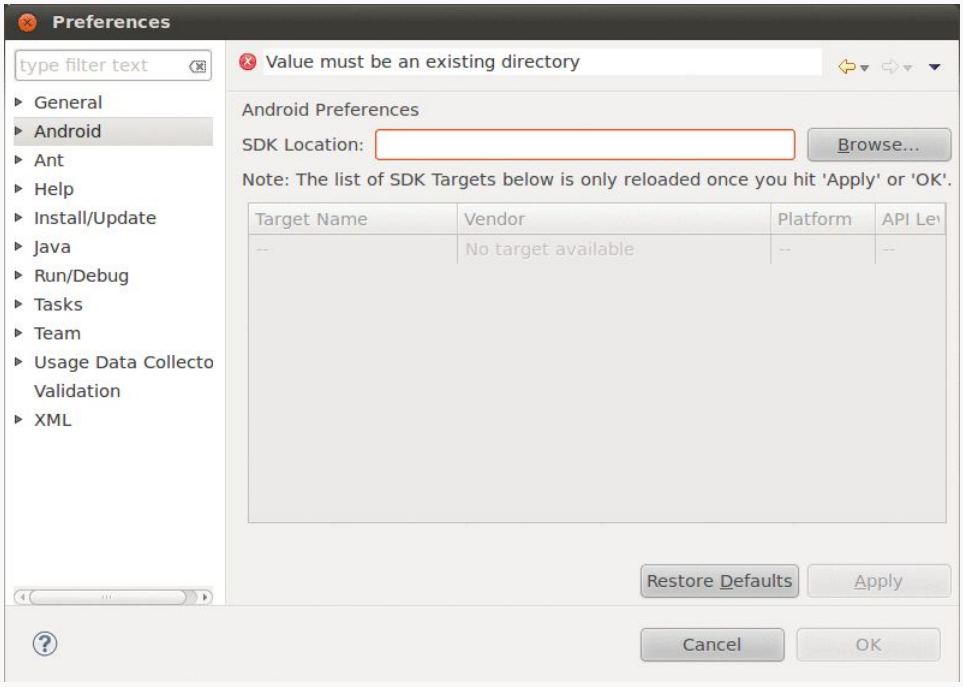


Figure 4: Eclipse ADT Configuration

Then, click the Browse... button to find the directory where you installed the SDK. After choosing it, click Apply on the Preferences window, and you should see the Android SDK versions you installed previously. Then, click OK, and the ADT will be ready for use.

Step #4 - Install the SDKs and Add-Ons

Next, we need to review what pieces of the Android SDK we have already and perhaps install some new items. To do that, you need to access the SDK Manager.

Android Studio First Run and SDK Manager Launch

When you first run Android Studio, you may be asked if you want to import settings from some other prior installation of Android Studio:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

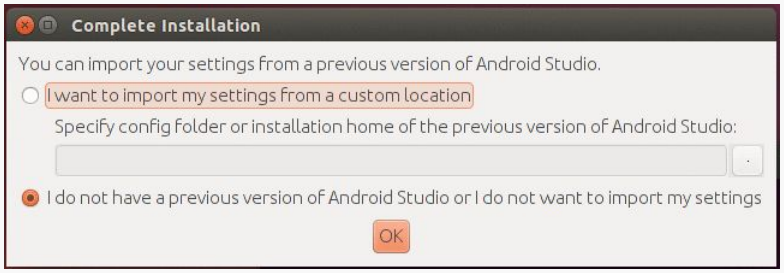


Figure 5: Android Studio First-Run Settings Migration Dialog

For most users, particularly those using Android Studio for the first time, the "I do not have..." option is the correct choice to make.

Then, after a short splash screen, you will be taken to the Android Studio Setup Wizard:

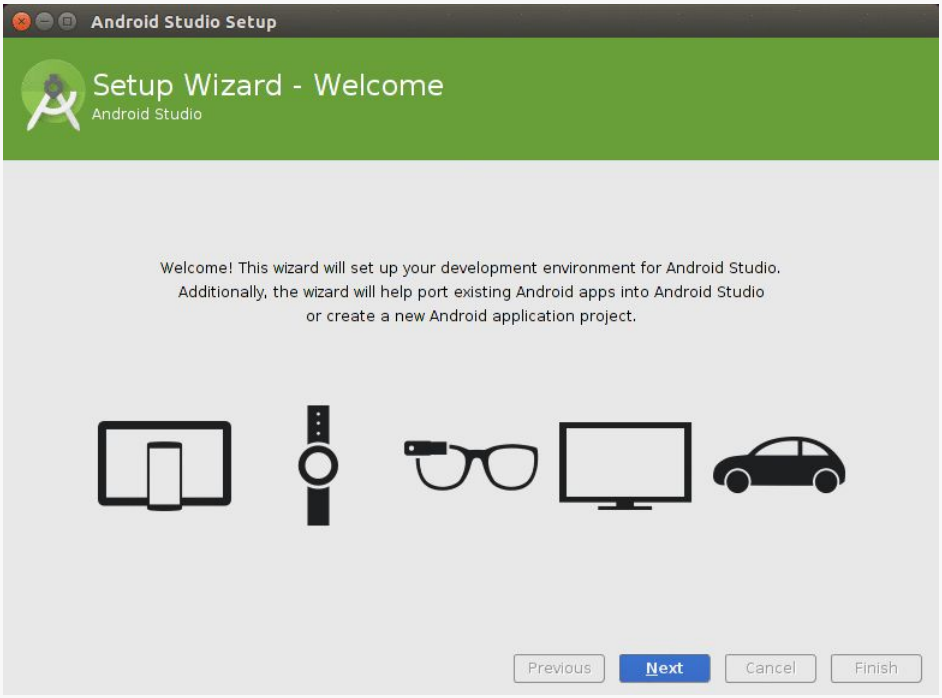


Figure 6: Android Studio Setup Wizard, First Page

Just click "Next" to advance to the second page of the wizard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

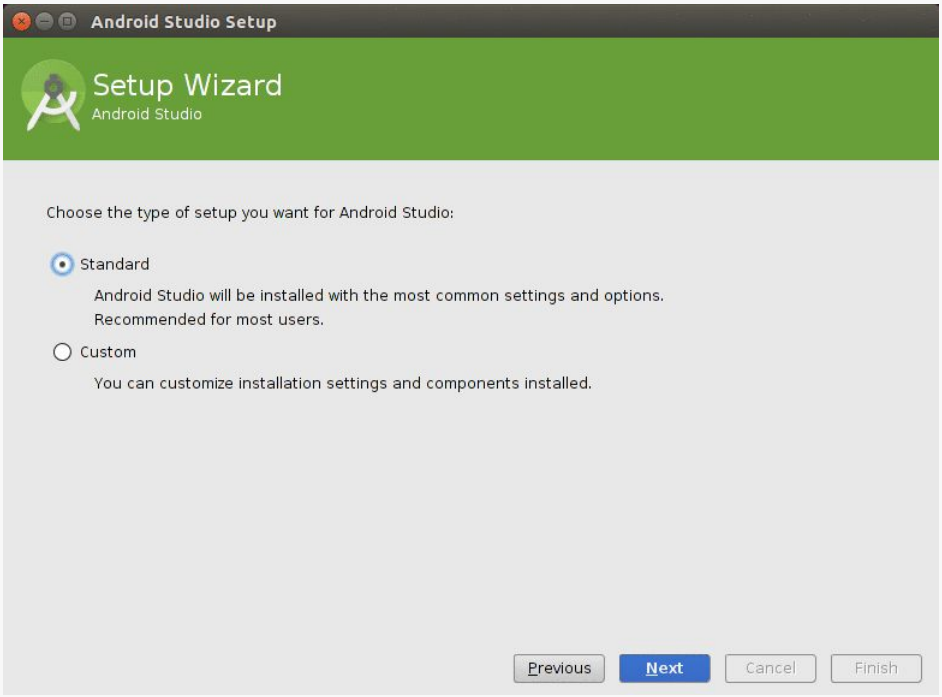


Figure 7: Android Studio Setup Wizard, Second Page

Here, you have a choice between "Standard" and "Custom" setup modes. "Custom" simply allows you to indicate what should be set up:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

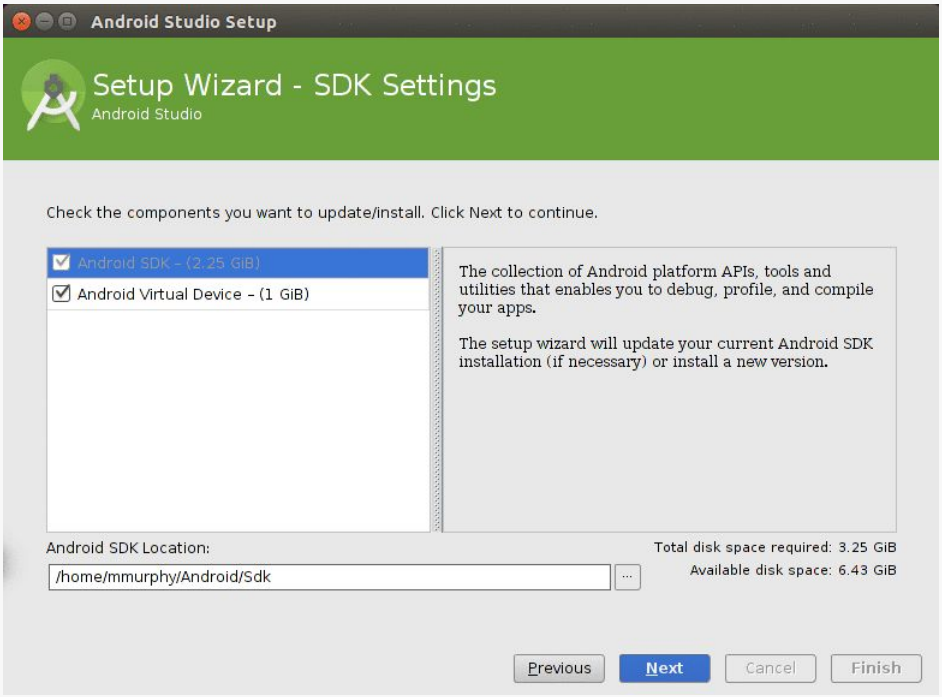


Figure 8: Android Studio Setup Wizard, Custom Page

Most likely, right now, you need all of that anyway, so the "Standard" route would be fine.

If you go the "Standard" route and click "Next", you may be taken to a wizard page explaining some information about the Android emulator:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

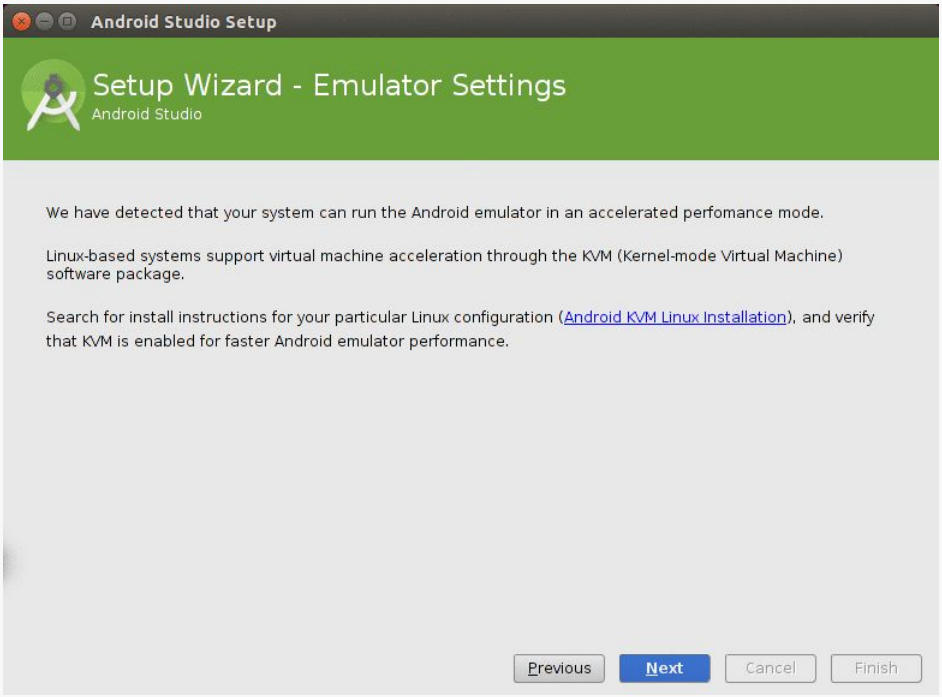


Figure 9: Android Studio Setup Wizard, Emulator Info Page

What is explained on this page may not make much sense to you. That is perfectly normal, and we will get into what this page is trying to say later in the book. Just click "Next" to advance to the next page where you will need to review the various licenses and click "Accept" if you wish to continue. At this point, clicking "Finish" will begin the setup process. This will include downloading a copy of the Android SDK and installing it into a directory adjacent to where Android Studio itself is installed.

When that is done, you will be taken to the Android Studio Welcome dialog:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

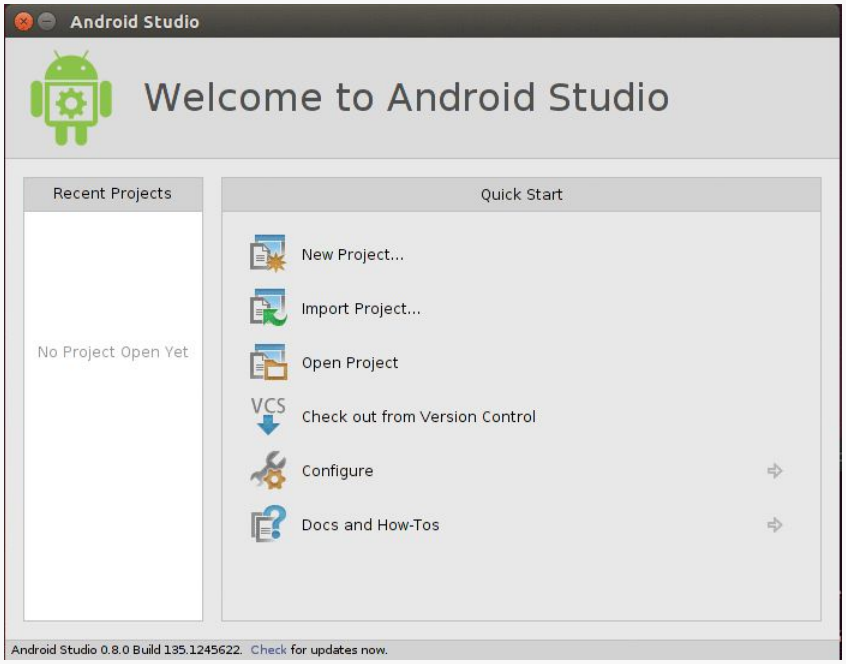


Figure 10: Android Studio Welcome Dialog

In very tiny print at the bottom of that dialog is a "Check for updates now" link. Click that, and if there are updates available, install them. This will automatically restart Android Studio. Android Studio *should* have downloaded the latest updates as part of the initial setup, so most likely this will indicate that nothing more is needed.

Then, in the welcome dialog, click Configure, to bring up a configuration sub-menu:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 11: Android Studio Welcome Dialog, Configure Sub-Menu

There, tap on SDK Manager to bring up the SDK Manager.

Eclipse SDK Manager Launch

Eclipse may take a few extra moments on its first launch to get going, but it does not do anything especially unusual.

Then, choose Windows > Android SDK Manager from the main menu, or tap on the toolbar icon that looks like the Android "bugdroid" mascot peeking out of the top of a box with a downward-pointing white arrow.

Using SDK Manager and Updating Your Environment

You should now have the SDK Manager window open:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

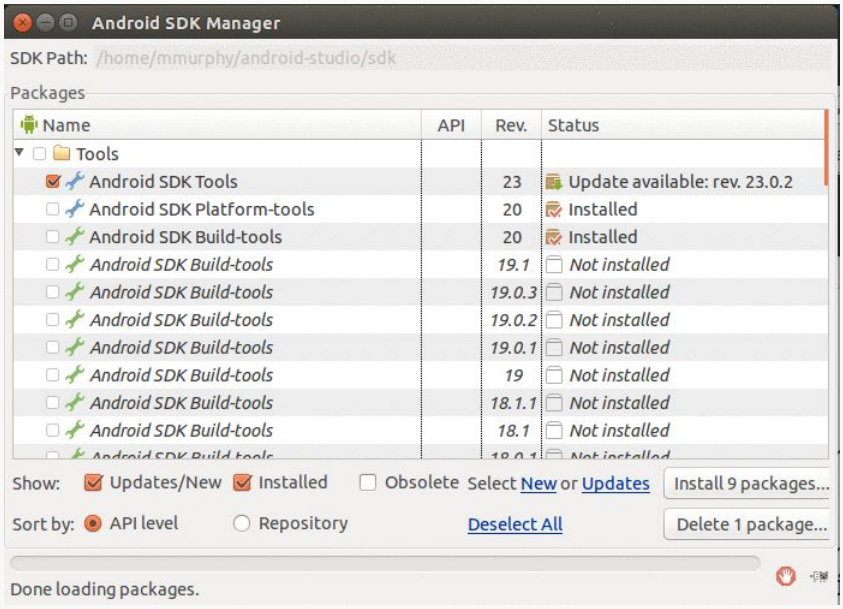


Figure 12: Android SDK Manager

At this point, while you have some of the build tools, you may lack the Java files necessary to compile an Android application. You also lack a few additional build tools, plus the files necessary to run an Android emulator. The checkboxes indicate which packages you want to install — by default, it pre-checks a number of them. If you chose the "ADT Bundle", some things will already be pre-installed for you.

You will want to install the following items, if they have not already been installed:

1. Android SDK Tools, Platform-tools, and the latest Build-tools.
2. "SDK Platform" for all Android SDK releases you want to code against — for this version of this book API 19 (Android 4.4) is recommended, along with any others with which you wish to experiment.
3. "ARM EABI v7a System Image", if there is an option for that for the API level you chose. You can also download the "Intel x86 Atom System Image", if one is available to you, as it is much faster, though setting that up is a bit of an advanced topic. You are also welcome to download similar images for any other Android API level that you are interested in testing against. However, for the purposes of this book, **DO NOT** choose the Android 4.4 or 4.4W emulator images, as the Android 4.4 emulator has a bug and the Android 4.4W emulator is only for Android Wear devices.
4. "Documentation for Android SDK" for the latest Android SDK release.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

5. "Samples for SDK" for the Android SDK release you chose in item #2 above, and perhaps for older releases if you wish.
6. Android Support Library and the Android Support Repository (in the Extras group at the bottom of the tree).

If you are running Windows, also choose the Google USB Driver (in the Extras group at the bottom of the tree).

Also, if anything that you presently have installed has updates available, they should already be pre-checked and will be updated when you install the items that you are adding.

Then, click the Install button beneath the tree on the right, which brings up a license confirmation dialog:

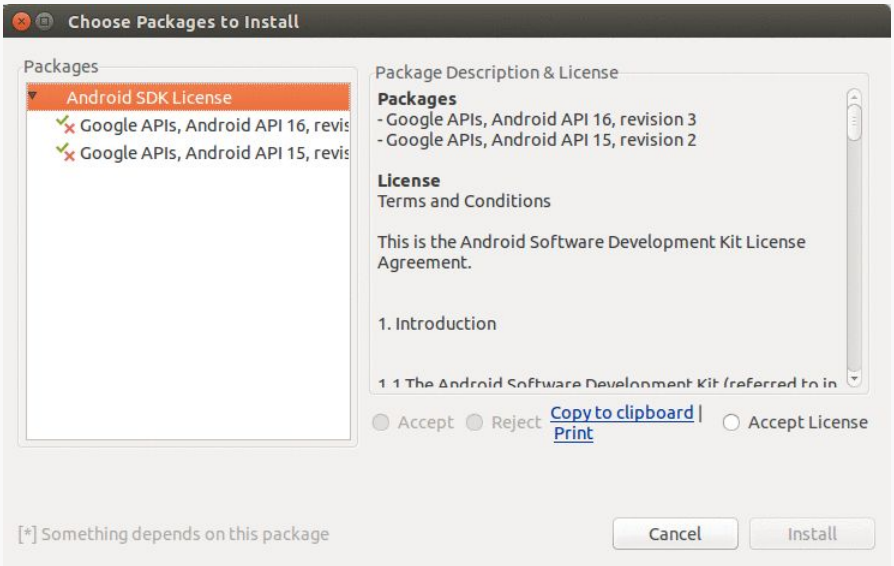


Figure 13: Android SDK Manager Installing Packages

Review and accept the licenses, then click the Install button.

When the download is complete, you can close up the SDK Manager.

In Our Next Episode…

… we will create an Android project that will serve as the basis for all our future tutorials, plus set up our emulator and device.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android and Projects

When you work on creating an app for Android, you will do so by working in a "project". The project is a directory containing your source code and other files, like images and UI definitions. Your IDE or other build tools will take what is in your project and generate an Android app (APK) as output.

The details of how you get started with a project vary based upon what IDE you are using, so this chapter goes through the various possibilities.

Common Concepts

The various ways we set up Android projects have some common elements.

The "Application Name" is the initial name of your project as seen by the user, in places like your home screen launcher icon and the list of installed applications.

The "Project Name" is the name of the project as it is represented inside of the IDE. As you type in an application name, the project name will automatically be filled in to match the application name, with whitespace and other invalid characters removed. Of course, you can change this as you see fit. In the case of Android Studio, the project name also forms the name of the directory that will hold the project.

The "Package Name" refers to a Java package name (e.g., `com.commonsware.empublite`). This package name will be used for generating some Java source code, and it also is used as a unique identifier of this package, as was mentioned earlier in this book.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The “Minimum Required SDK” refers to how far back in Android’s version history you are willing to support. The lower the value you specify here, the more Android devices can run your app, but the more work you will have to do to test whether your app really does support those devices. Nowadays, for new development, a minimum required SDK of 15 is reasonable, and you can change your chosen value later on if needed.

The “Target SDK”, roughly speaking, is the version of Android you were thinking of when you were writing the code for this app. Usually, you will set this to be the latest shipping Android API level, then change it over time as new versions of Android are released and you decide that you are ready for some of those changes. We will be exploring the ramifications of target SDK versions throughout the book.

The “Compile With” (a.k.a., “build SDK” or `compileSdkVersion`) is the version of Android whose classes and methods you want to compile against. This can be newer than the minimum required SDK, and it often is newer. On newer devices running newer versions of Android, you might want to take advantage of some new features, and you will “route around” that code on older devices to maintain backwards compatibility. Hence, typically, your build SDK is set to a fairly new version of Android, certainly one new enough to support all of the classes and methods from the Android SDK that you want to use. Note that to set this to API Level 21 or higher, you will need to be using Java 7 or higher for your Java compiler.

The “Theme” is a general statement of the look and feel of your app, particularly in terms of color scheme. The current default (“Holo Light with Dark Action Bar”) means that the body of your UI will be dark text on a light background, except for the “action bar” across the top, which will be light text on a dark background. You will be able to create your own custom themes, overriding various characteristics from one of these stock themes, to set up your own color scheme and the like. We will explore that process later in the book.

Projects and Android Studio

You may have chosen to use Android Studio as your IDE.

With Android Studio, to work on a project, you can either create a new project from scratch, you can copy an existing Android Studio project to a new one, or you can import an existing Android project into Android Studio. The following sections will review the steps needed for each of these.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Creating a New Project

You can create a project from one of two places:

- If you are at the initial dialog that you first encountered when you opened Android Studio, choose the "Create Project..." menu item
- If you are inside the Android Studio IDE itself, choose File > Create Project... from the main menu

This brings up the new-project wizard:

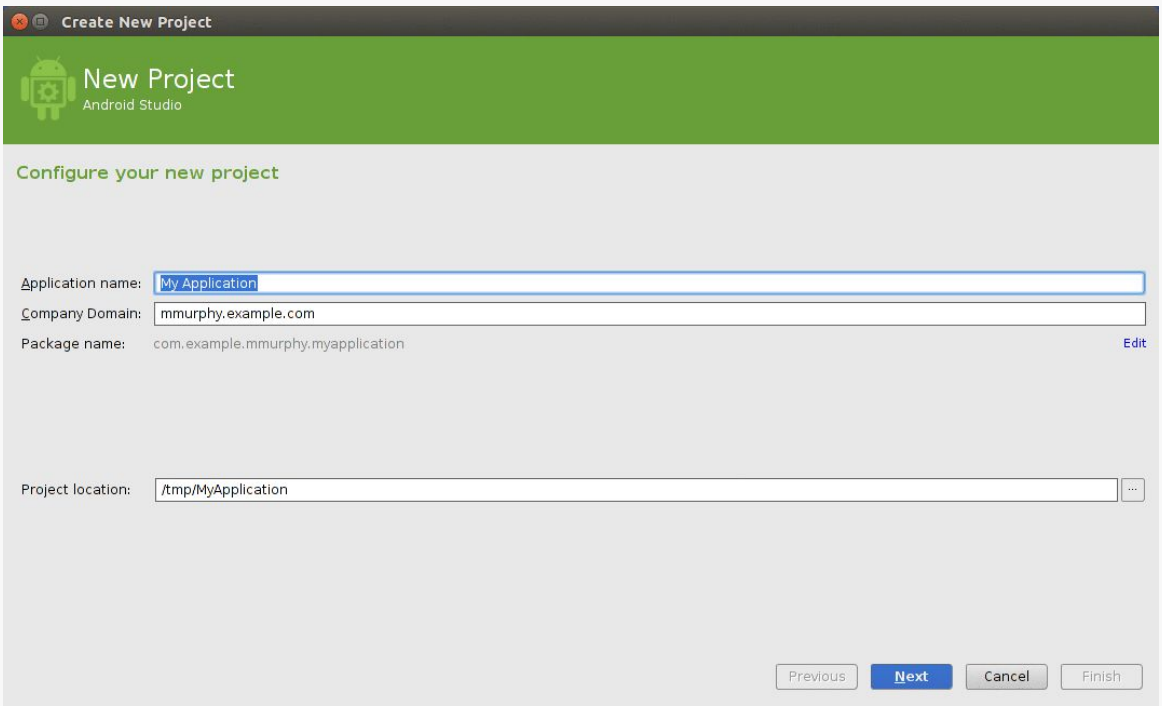


Figure 14: Android Studio Create-Project Wizard, First Page

The first page of the wizard is where you can specify:

- The application name
- The package name
- The directory where you want the project files to go

By default, the package name will be made up of two pieces:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. The domain name that you specify in the “Company Domain” field
2. The application name, converted into all lowercase with no spaces or other punctuation

If this is not what you want, click the tiny “Edit” link on the far right side of the proposed package name, which will now allow you to edit the package name directly:

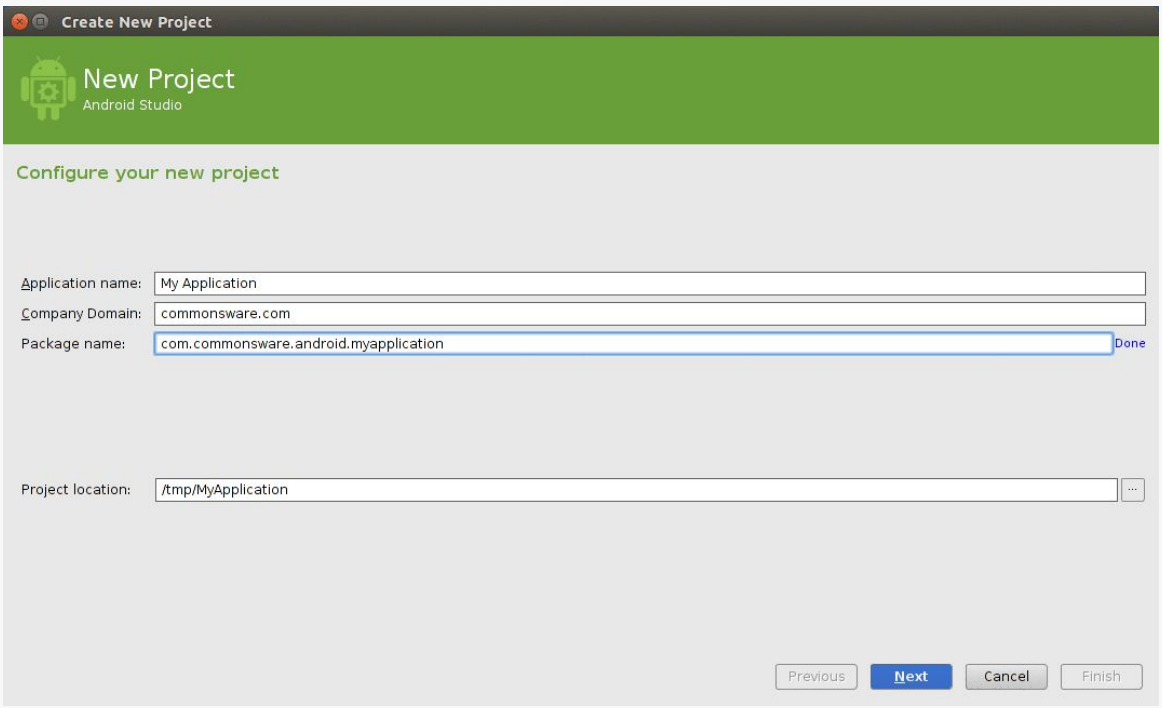


Figure 15: Android Studio Create-Project Wizard, First Page, with Editable Package Name

Clicking “Next” will advance you to a wizard page where you indicate what sort of project you are creating, in terms of intended device type (phones/tablets, TVs, etc.) and minimum required SDK level:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

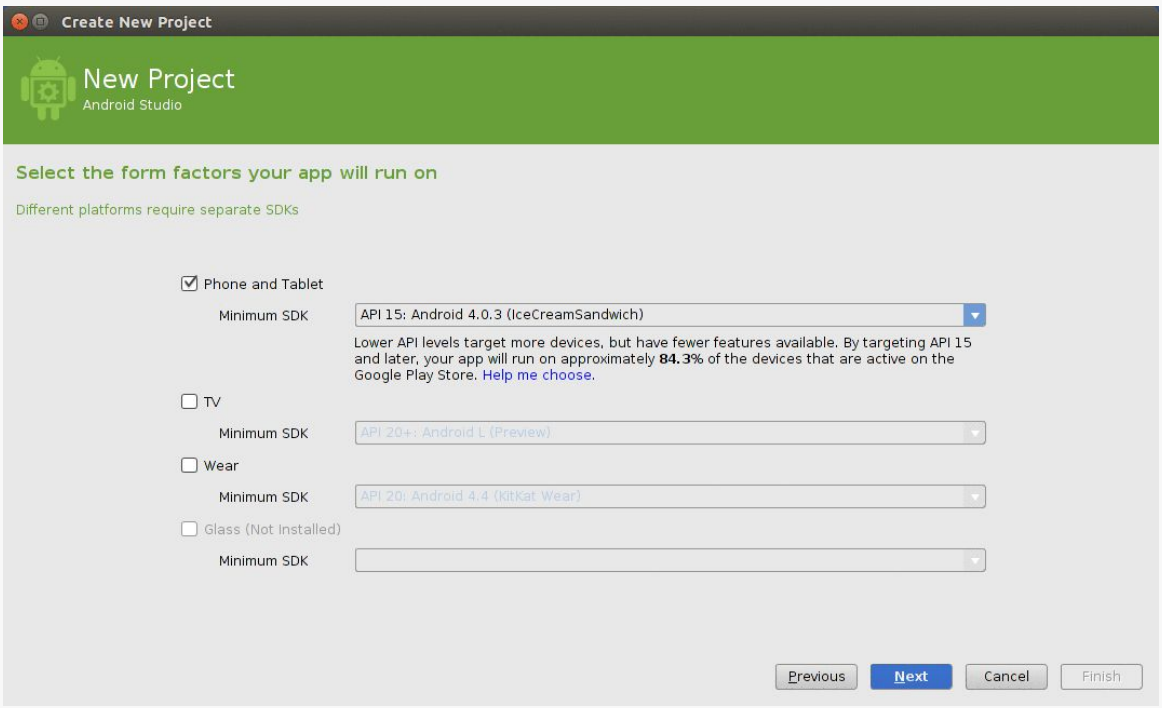


Figure 16: Android Studio Create-Project Wizard, Second Page

Developers just starting out on Android should only check "Phone and Tablet" as the device type.

Clicking "Next" advances you to the third page of the wizard, where you can choose if Android Studio should create an initial activity for you, and if so, based on what template:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

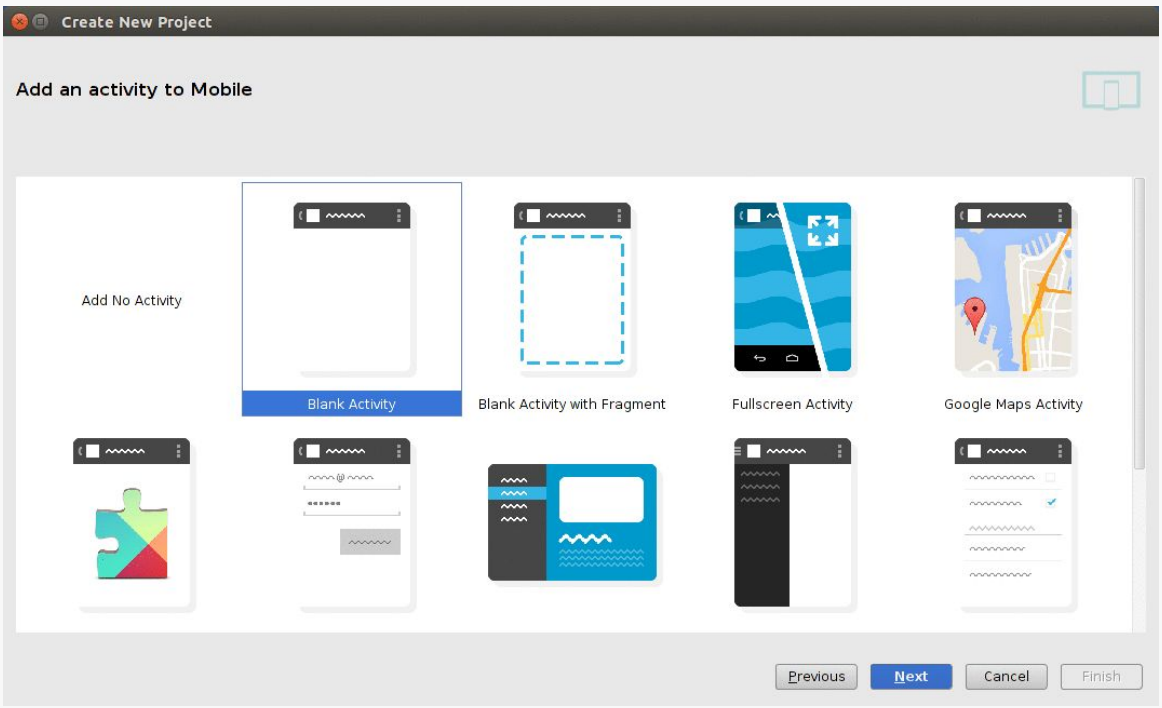


Figure 17: Android Studio Create-Project Wizard, Third Page

"Blank Activity" is a good choice for first-time Android developers.

If you choose any option other than "Add No Activity", clicking "Next" will advance you to a page in the wizard where you can provide additional details about the activity to be created:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

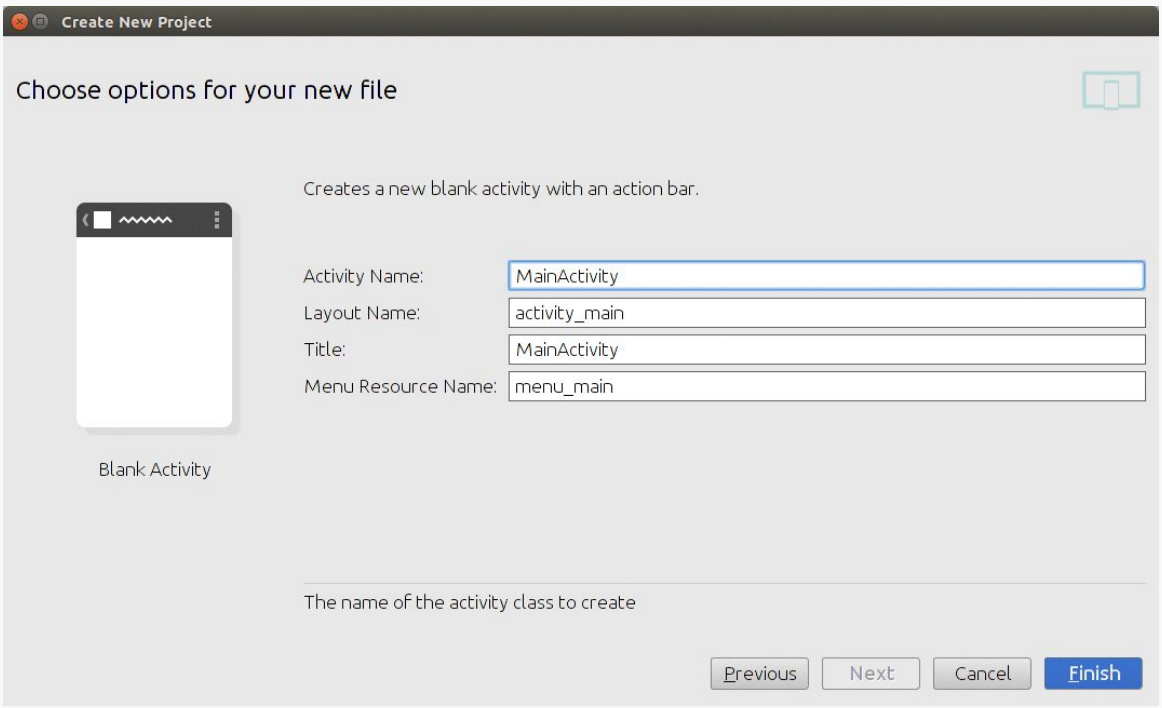


Figure 18: Android Studio Create-Project Wizard, Fourth Page

What options appear here will vary based upon the template you chose in the previous page. Common options include "Activity Name" (the name of the Java class for your activity), "Layout Name" (the base name of an XML file that will contain a UI definition of your activity), and "Title" (the caption that should appear at the top of your activity).

Clicking "Finish" will generate your project files.

Copying a Project

Android Studio projects are simply directories of files, with no special metadata held elsewhere (as is the case with Eclipse). Hence, to copy a project, just copy its directory.

Importing a Project

You can import a project from one of two places:

- If you are at the initial dialog that you first encountered when you opened Android Studio, choose the "Import Project..." menu item

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- If you are inside the Android Studio IDE itself, choose File > Import Project... from the main menu

Then, choose the directory containing the project to be imported.

What happens now depends upon the nature of the project. If the project was already set up for use with Android Studio, or at least with Gradle for Android, the Android Studio-specific files will be created (or updated) in the project directory.

However, if the project was not set up for Android Studio or Gradle for Android, but does have Eclipse project files (or at least a `project.properties` file), you will be led through an Eclipse import wizard.

The first page of that wizard is where you specify where Android Studio should make a copy of the project, so it does not modify anything with the original directory:

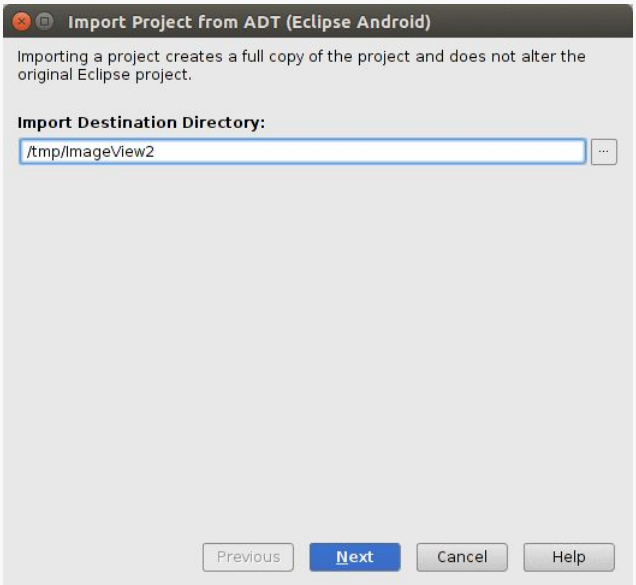


Figure 19: Android Studio Eclipse Import Wizard, First Page

Clicking "Next" will bring up a page where you can configure some automatic fixes that the import wizard will apply to the imported project code. The details of what is going on here are well past what we have covered so far in the book. Normally, the defaults are fine.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

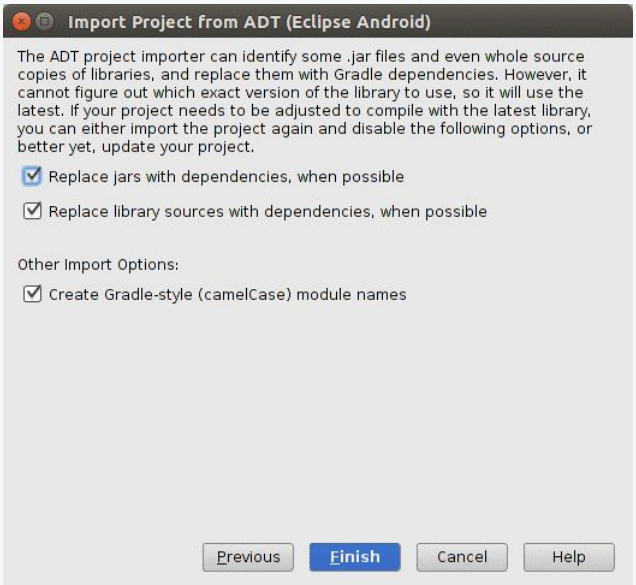


Figure 20: Android Studio Eclipse Import Wizard, Second Page

Clicking "Finish" will perform the project conversion. Android Studio will open up an `import-summary.txt` file outlining some details of how the conversion was accomplished. At this point, the copied-and-modified project is ready for use.

Projects and Eclipse

You may have chosen to use Eclipse with the ADT plugin as your IDE, whether based on the downloadable ADT Bundle or by adding the ADT plugin and Android SDK to an existing copy of Eclipse.

As with Android Studio, to work on a project in Eclipse, you can either create a new project from scratch, you can copy an existing Eclipse project to a new one, or you can import an existing Android project into Eclipse. The following sections will review the steps needed for each of these.

Creating a New Project

From the Eclipse main menu, choose File > New > Project... to bring up the first page of the "New Project" wizard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

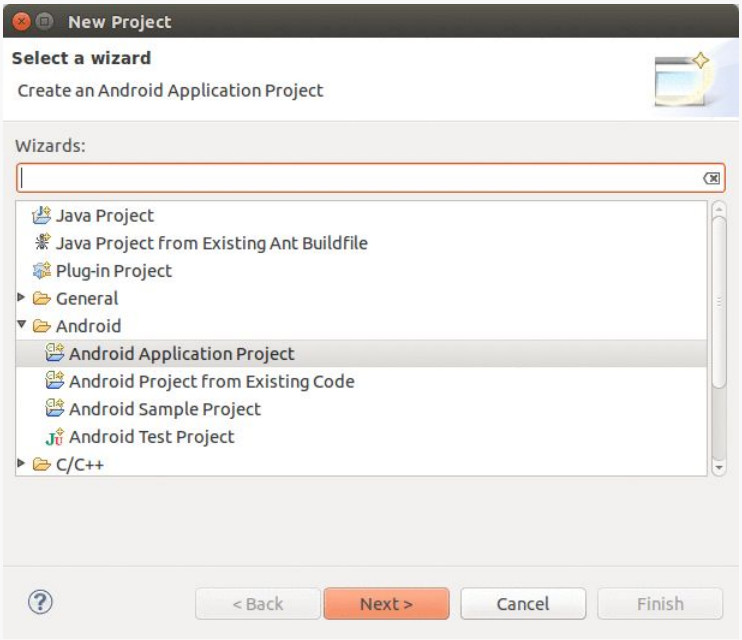


Figure 21: Eclipse New Project Wizard

Choose "Android Application Project" from the types of projects and click "Next >" to proceed to the next page of the wizard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

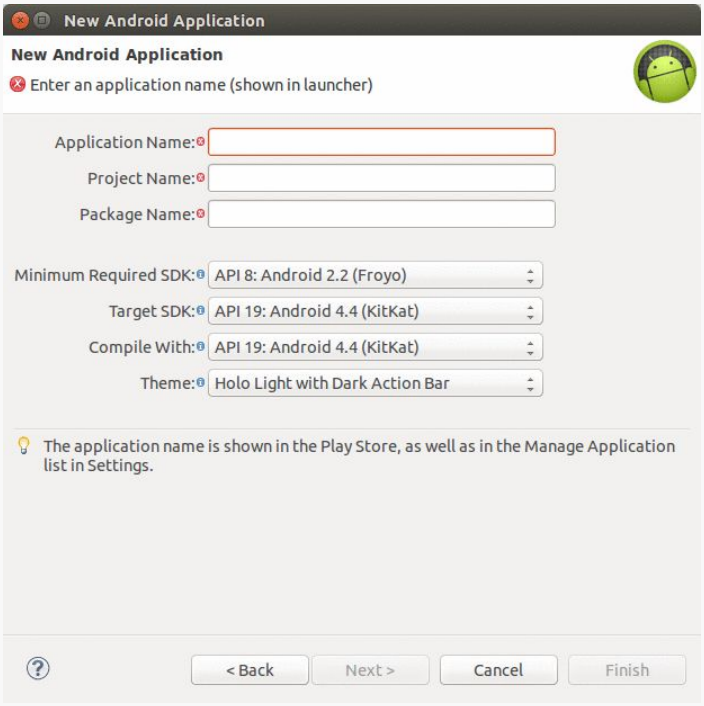


Figure 22: Eclipse New Android Application Project Wizard, As Initially Launched

The elements on this wizard page are described in the "Common Elements" section that opened this chapter.

Once those are filled in, you can click "Next >" to move to the next page of the wizard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

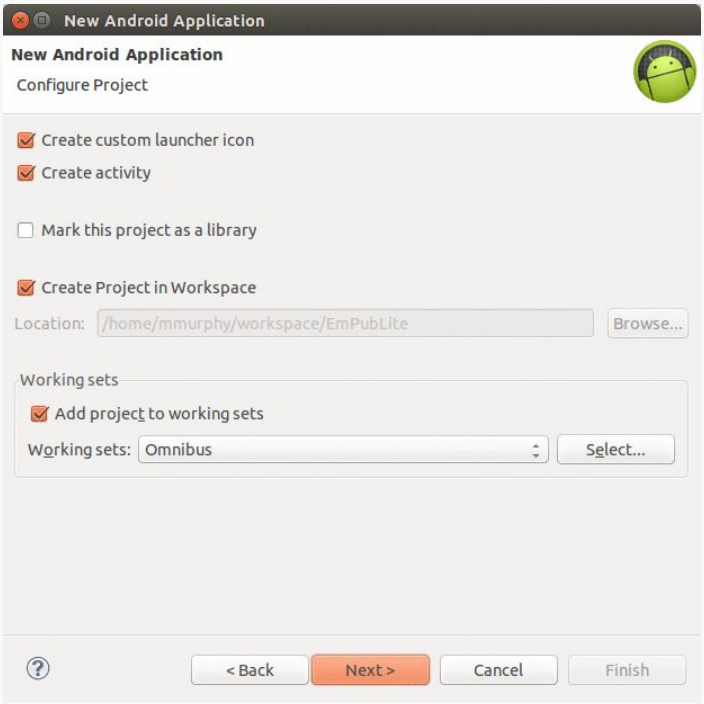


Figure 23: Eclipse Wizard, Other Project Settings, As Initially Launched

The "Create custom launcher icon" checkbox is asking you if you want to customize your app's icon right now or not. You can always do this later, so for your first few projects, it is simplest if you uncheck this option. We will explore how to customize your app's icon later in this book. If you leave this unchecked, the new-project wizard will give you a stock icon at the outset.

The "Create activity" checkbox indicates if you want the new-project wizard to also create your first "activity" for you — in other words, creating the first screen of your UI. You do not have to have the new-project wizard create this for you, as you can add activities later on yourself, whether through a separate new-activity wizard or by just creating the appropriate files yourself. Hence, whether you leave this checked or not is up to you.

The "Mark this project as a library" checkbox indicates whether you are building an app or a library to be used by other apps. Most times, you will be creating an app, and so you will leave this checkbox unchecked. We will see what Android library projects are and what they mean later in the book.

The remaining options on this wizard page are for Eclipse:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Choose where you want the project files to be placed, either by leaving "Create Project in Workspace" checked, or unchecking it and choosing a directory on your development machine in which to place the files
- If you are using Eclipse's working sets, choose your working set (if you do not know what working sets are in Eclipse, you are not using them, and so you can safely ignore this option)

Once you have filled in this page of the wizard, there are three possible next steps:

1. If you left the "Create custom launcher icon" item checked, clicking "Next >" will take you to a wizard page, from the new-icon-set wizard, for creating your launcher icon. We will explore that wizard later in this book.
2. If you unchecked both the "Create custom launcher icon" and the "Create activity" checkboxes, the "Finish" button will be enabled, and clicking it will create your project.
3. If you left the "Create activity" checkbox checked, the "Next >" button will take you to a wizard page, from the new-activity wizard, for creating your first activity:

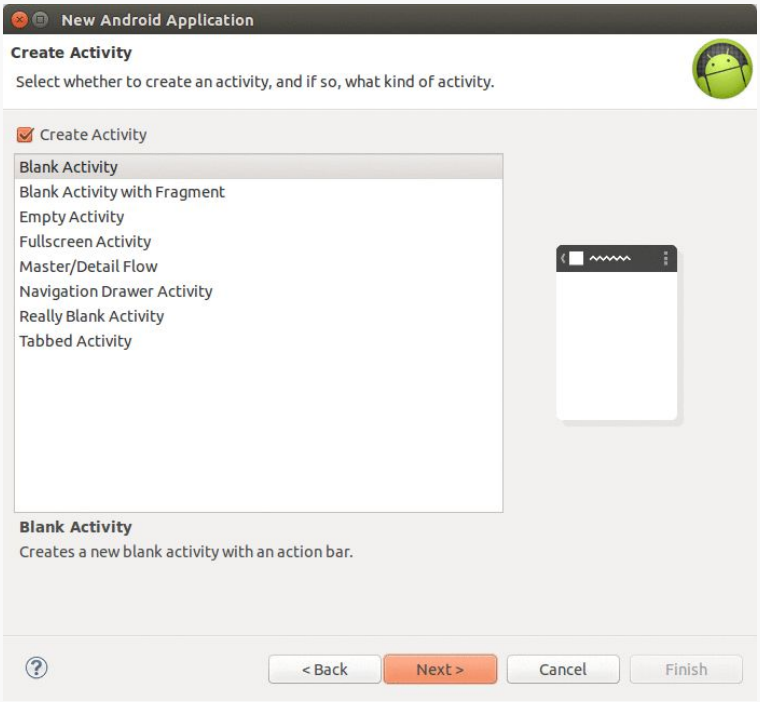


Figure 24: Eclipse New Android Project Wizard, Create Activity Page

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here, you choose which template activity you want to use as a starting point. The exact roster of templates will vary based upon what is shipping with Android and what third-party templates you may have installed. "Empty Activity" generates the least code for you; everything else adds more stuff that you can use (if you want) or rip out (if you did not want it).

Clicking "Next >" will bring up a wizard page where you can fill in details to be used by your chosen activity template:

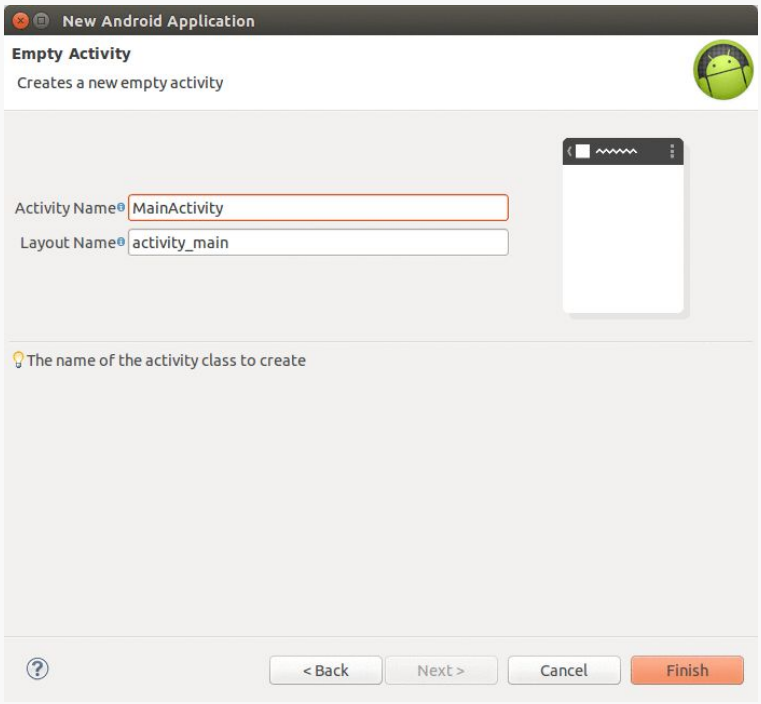


Figure 25: Eclipse New Android Project Wizard, New Empty Activity Page

At minimum, the wizard page should ask you for an "Activity Name" (the name of the Java class for your activity) and the "Layout Name" (the base name of an XML file that will contain a UI definition of your activity). Other fields, drop-down lists, and so forth may be available depending upon the particular template that you chose.

Once you click the "Finish" button on the wizard, the wizard will generate your project for you and take you to it in Eclipse's Package Explorer.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Copying a Project

If you already have a project in Eclipse that you just want to copy and use as a starting point for a new project, just right-click over the project in the Eclipse Package Explorer and choose "Copy" from the context menu. Or, use your platform's standard hot-key for the copy operation (e.g., `<Ctrl>-<C>` on Windows and Linux).

Then, you can paste in the project to the Package Explorer, either through Edit > Paste in the Eclipse main menu or through a hot-key (e.g., `<Ctrl>-<V>` on Windows and Linux). This will bring up a "Copy Project" Eclipse dialog:

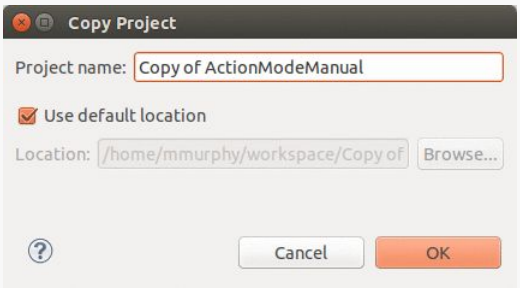


Figure 26: Eclipse Copy Project Dialog

Here, you can choose the name of the project copy and where to copy its files to on your development machine. After clicking OK, Eclipse will copy the project.

Importing a Project

It may be that you have some source code to an Android project, such as from a public open source project, that you want to load into Eclipse. Eclipse has a means of importing such projects, and we will see how to do that in the next tutorial.

Starter Project Generators

In addition to creating projects through an IDE's new-project wizard, there are various Web sites that offer online project generators:

- Android Bootstrap
- Android Kickstartr

On those sites, you provide basic configuration data, such as your application's package name, and they generate a complete starter project for you. These projects

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

tend to be significantly more advanced than what you get from the IDE wizards. On the plus side, you get a more elaborate "scaffold" on which you can "hang" your own business logic. However, understanding what those generators create and how to change the generated code requires a fair bit of Android development experience.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #2 - Creating a Stub Project

Creating an Android application first involves creating an Android "project". As with many other development environments, the project is where your source code and other assets (e.g., icons) reside. And, the project contains the instructions for your tools for how to convert that source code and other assets into an Android APK file for use with an emulator or device, where the APK is Android's executable file format.

Hence, in this tutorial, we kick off development of a sample Android application, to give you the opportunity to put some of what you are learning in this book in practice.

About Our Tutorial Project

The application we will be building in these tutorials is called `EmPubLite`. `EmPubLite` will be a digital book reader, allowing users to read a digital book like the one that you are reading right now.

`EmPubLite` will be a partial implementation of the `EmPub` reader used for the APK version of this book. `EmPub` itself is a fairly extensive application, so `EmPubLite` will have only a subset of its features. The main `EmPub` app, however, will be used elsewhere in this book to illustrate more advanced Android capabilities.

The "Em" of `EmPub` and `EmPubLite` stands for "embedded". These readers are not designed to read an arbitrary EPUB or MOBI formatted book that you might download from somewhere. Rather, the contents of the book (largely an unpacked EPUB file) will be "baked into" the reader APK itself, so by distributing the APK, you are distributing the book.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

About the Rest of the Tutorials

Of course, you may have little interest in writing a digital book reader app.

The tutorials presented in this book are certainly optional. There is no expectation that you have to write any code in order to get value from the book. These tutorials are here simply as a way to help those of you who "learn by doing" have an opportunity to do just that.

Hence, there are any number of ways that you can use these tutorials:

- You can ignore them entirely. That is not the best answer, but you are welcome to do it.
- You can read the tutorials but not actually do any of the work. This is the best low-effort answer, as it is likely that you will learn things from the tutorials that you might have missed by simply reading the non-tutorial chapters.
- You can follow along the steps and actually build the `EmPubLite` app.
- You can download the answers from the book's GitHub repository. There, you will find one directory per tutorial, showing the results of having done the steps in that tutorial. For example, you will find a `T2-Project/` directory containing a copy of the `EmPubLite` sample app after having completed the steps found in this tutorial. There are separate directories for Android Studio (`EmPubLite-AndroidStudio/`) and Eclipse (`EmPubLite/`) editions of the projects. You can import these projects into your IDE, examine what they contain, cross-reference them back to the tutorials themselves, and run them.

Any of these are valid options — you will need to choose for yourself what you wish to do.

About Our Tools

The instructions in the remaining tutorials should be accurate for:

- Android Studio 1.0
- Version 23.0.x of the Eclipse ADT plugin

The instructions *may* work for other versions of these IDEs, but there may also be some differences.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Step #1: Creating the Project

We need to create the Android project for `EmPubLite`.

First, visit the book's GitHub repository's "releases" area and download the ZIP file that corresponds with this version of the book and the tools that you wish to use:

- For Android Studio, download `EmPubLiteStarter-AndroidStudio.zip`
- For Eclipse, download `EmPubLiteStarter-Eclipse.zip`

Then, unZIP that ZIP archive into some directory on your development machine outside of where your IDE resides. The ZIP archive will expand into a set of files and subdirectories. Most likely, you will want to place those into an empty existing directory. Eclipse users should **NOT** unZIP the archive directly into the Eclipse workspace, as Eclipse gets confused easily.

Many steps in the tutorials have the same basic instructions regardless of IDE or toolchain that you wish to use. Some steps, though, have tool-specific instructions.

This step has two sets of instructions:

1. Importing the starter project into Android Studio
2. Importing the starter project into Eclipse

Just click on the link to jump to the set of instructions that you wish to follow.

Android Studio Project Import

You can import a project from one of two places:

- If you are at the initial "welcome" dialog that you first encountered when you opened Android Studio, choose the "Import Non-Android Studio Project..." menu item
- If you are inside the Android Studio IDE itself, choose File > Import Project... from the main menu

Then, just choose the `EmPubLite/` directory inside where you unzipped `EmPubLiteStarter-AndroidStudio.zip`.

Since this project is already set up for use with Android Studio, you should be taken right into the main IDE.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio has two ways of viewing the contents of Android projects. The default one, that you are presented with when importing the project, is known as the "Android project view":

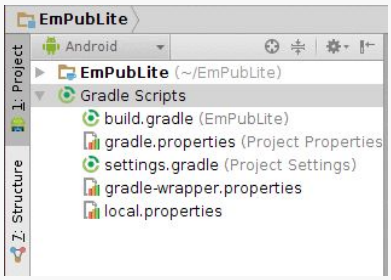


Figure 27: Android Studio "Android Project View"

While you are welcome to navigate your project using it, the tutorial chapters in this book, where they have screenshots of Android Studio, will show the classic project view:

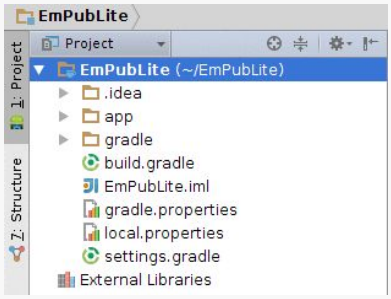


Figure 28: Android Studio "Classic Project View"

To switch to this classic view — and therefore match what the tutorials will show you — click on the drop-down list that currently shows "Android" as selected and choose "Project" instead.

Eclipse Project Import

In Eclipse, choose File > Import from the main menu, to bring up the import wizard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

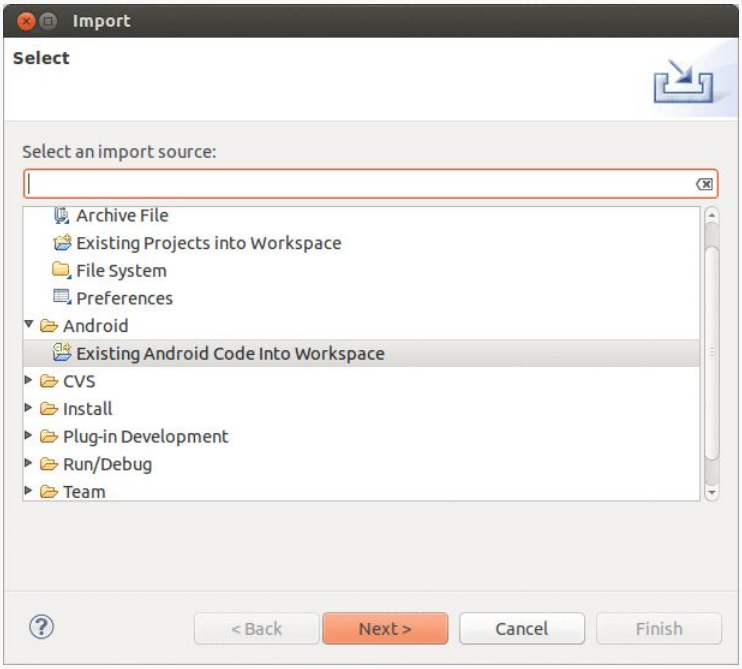


Figure 29: Eclipse Import Wizard, First Page

Choose "Existing Android Code Into Workspace" and click the Next button to bring up the next page of the wizard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

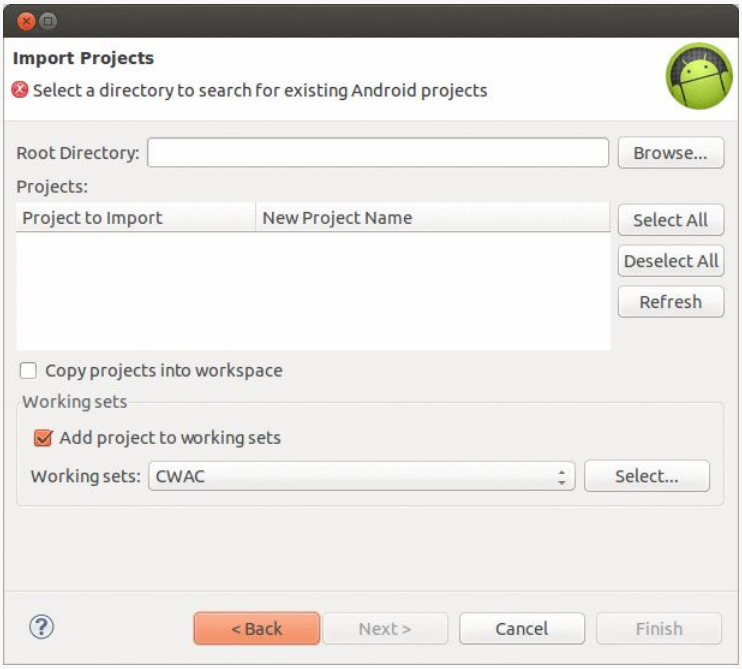


Figure 30: Eclipse Import Wizard, Second Page

Click the "Browse..." button towards the upper-right of the wizard, and from there choose the directory containing the unZIPped version of `EmPubLiteStarter-Eclipse.zip`.

This will change the wizard to show the project. Click on "EmPubLiteActivity" in the table, which will then become editable. Change `EmPubLiteActivity` to be `EmPubLite`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

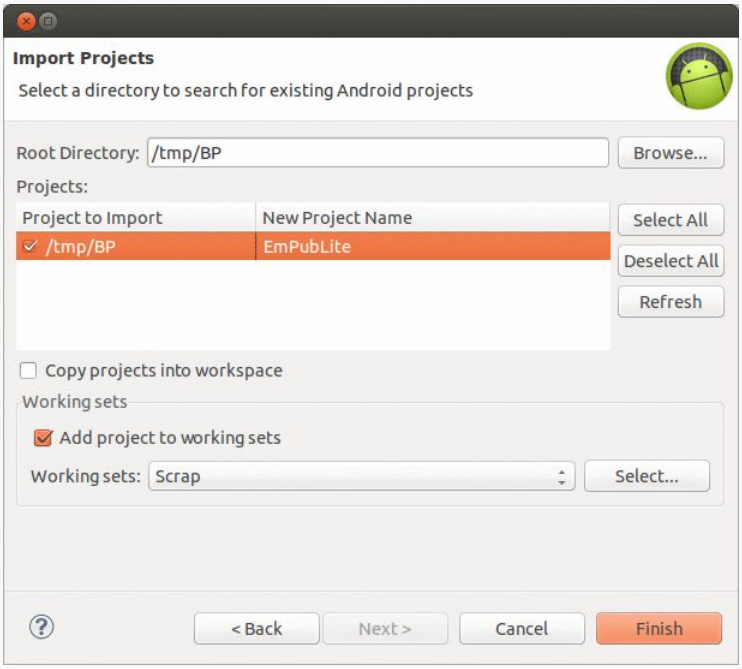


Figure 31: Eclipse Import Wizard, Second Page, After Directory Selected

If you want to have the contents of your chosen directory copied into your Eclipse workspace, click the "Copy projects into workspace" checkbox. Otherwise, Eclipse will keep the project files where they are.

If you are using Eclipse's working sets, choose your working set (if you do not know what working sets are in Eclipse, you are not using them, and so you can safely ignore this option).

Then, click Finish. Afterwards, you should be able to find EmPubLite in the Eclipse Package Explorer:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

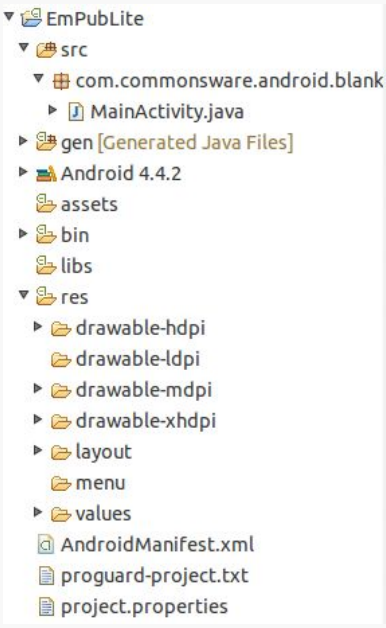


Figure 32: Eclipse Package Explorer, Showing EmPubLite from Import

You may encounter some problems, if the imported project is set to build with an Android SDK that you have not installed. To check your settings for this, right-click over the EmPubLite entry in the Package Explorer, and choose Properties. In the Properties dialog, click the Android entry in the list on the left:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

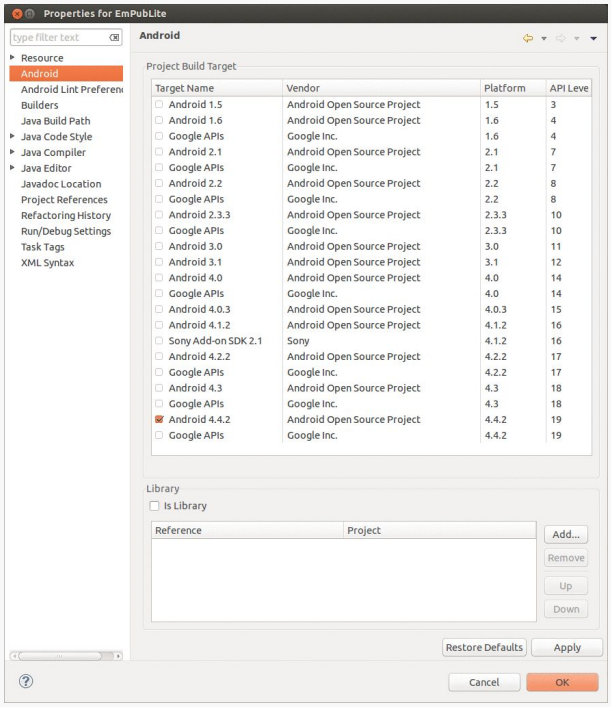


Figure 33: Eclipse Project Properties

If none of the entries in the API level checklist are checked, check the highest one that you have installed, if that is 15 or higher.

Step #2 - Set Up the Emulator

The Android tools include an emulator, a piece of software that pretends to be an Android device. This is very useful for development — not only does it mean you can get started on Android without a device, but the emulator can help test device configurations that you do not own.

Your first decision to make is whether or not you want to bother setting up an emulator image right now. If you have an Android device, you may prefer to start testing your app on it, and come back to set up the emulator at a later point.

Your second decision to make is whether you want to go ahead and set up the x86 emulator support. The vast majority of Android devices have ARM CPUs, while the vast majority of development machines have x86 CPUs. The ARM emulator is *slow* on x86 machines, as every ARM instruction must be translated into corresponding x86 instruction(s) before it can be executed. However, setting up the x86 emulator

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

support is a bit complicated, and your development machine may not be able to support it. If you wish to try to set up the x86 emulator right away, there are instructions for doing that later in the book that you can review and follow.

The Android emulator can emulate one or several Android devices. Each configuration you want is stored in an "Android virtual device", or AVD. The AVD Manager is where you create these AVDs.

Android Studio now has its own implementation of the AVD Manager that is separate from the one Android developers have traditionally used.

AVD Manager in Android Studio

To open the AVD Manager in Android Studio, choose Tools > Android > AVD Manager from the main menu. You should see a table of available virtual devices (AVDs), possibly including one already set up for you:

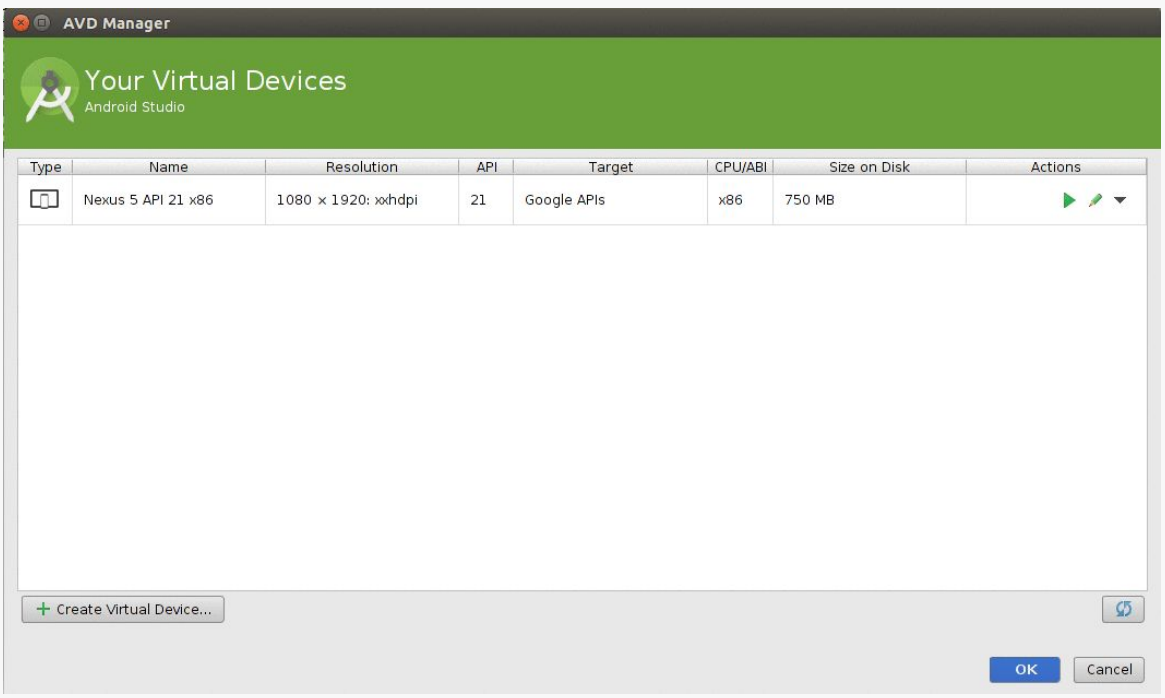


Figure 34: Android Studio AVD Manager, First Page

To define a new AVD, click the "Create Virtual Device" button in the AVD Manager, which brings up a "Virtual Device Configuration" wizard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

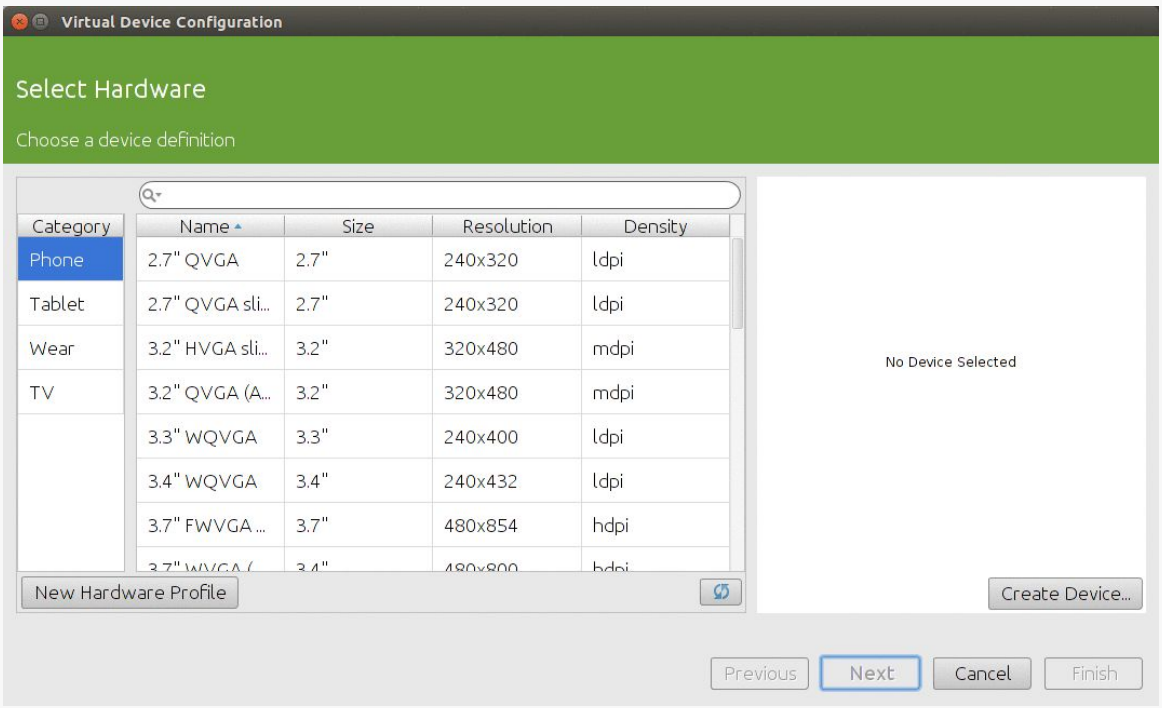


Figure 35: Android Studio Virtual Device Configuration Wizard, First Page

The first page of the wizard allows you to choose a device profile to use as a starting point for your AVD. The "New Hardware Profile" button allows you to define new profiles, if there is no existing profile that meets your needs.

Since emulator speeds are tied somewhat to the resolution of their (virtual) screens, you generally aim for a device profile that is on the low end but is not completely ridiculous. For example, an 800x480 phone would be considered by many people to be fairly low-resolution. However, there are plenty of devices out there at that resolution (or lower), and it makes for a reasonable starting emulator.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

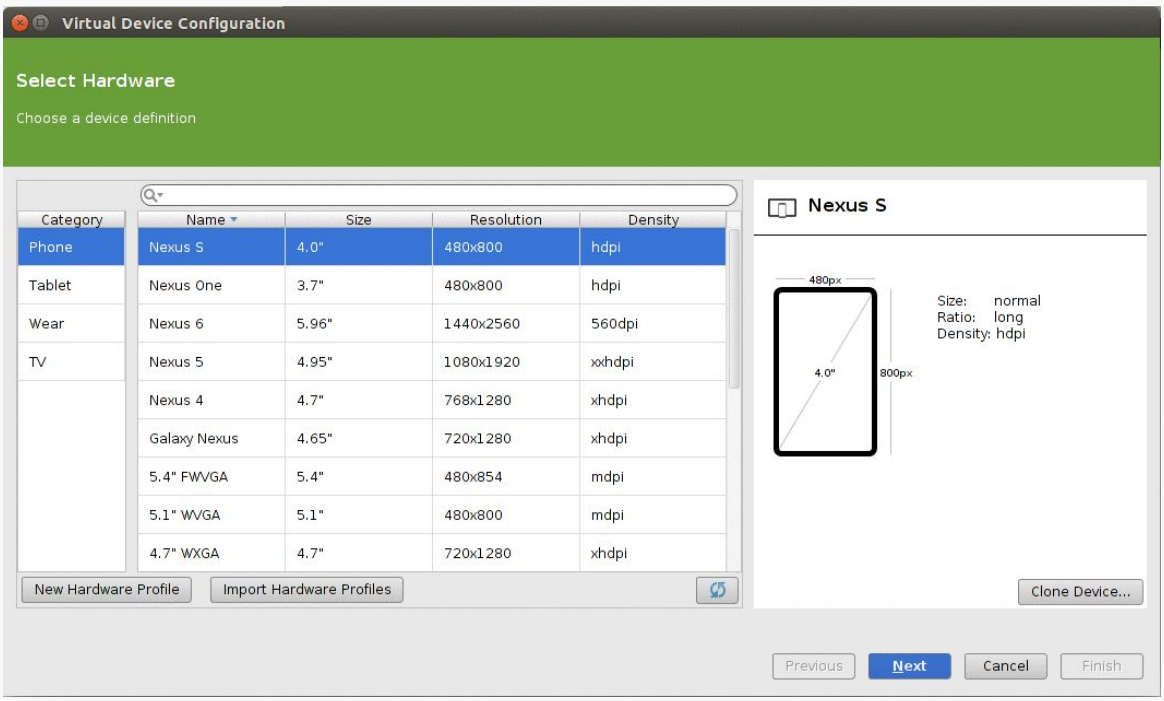


Figure 36: Android Studio Virtual Device Configuration Wizard, First Page, Nexus S Selected

If you want to create a new device profile based on an existing one — to change a few parameters but otherwise use what the original profile had – click the "Clone Device" button once you have selected your starter profile.

However, in general, at the outset, using an existing profile is perfectly fine.

Clicking "Next" allows you to choose an emulator image to use, based on the ones that you downloaded using the SDK Manager:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

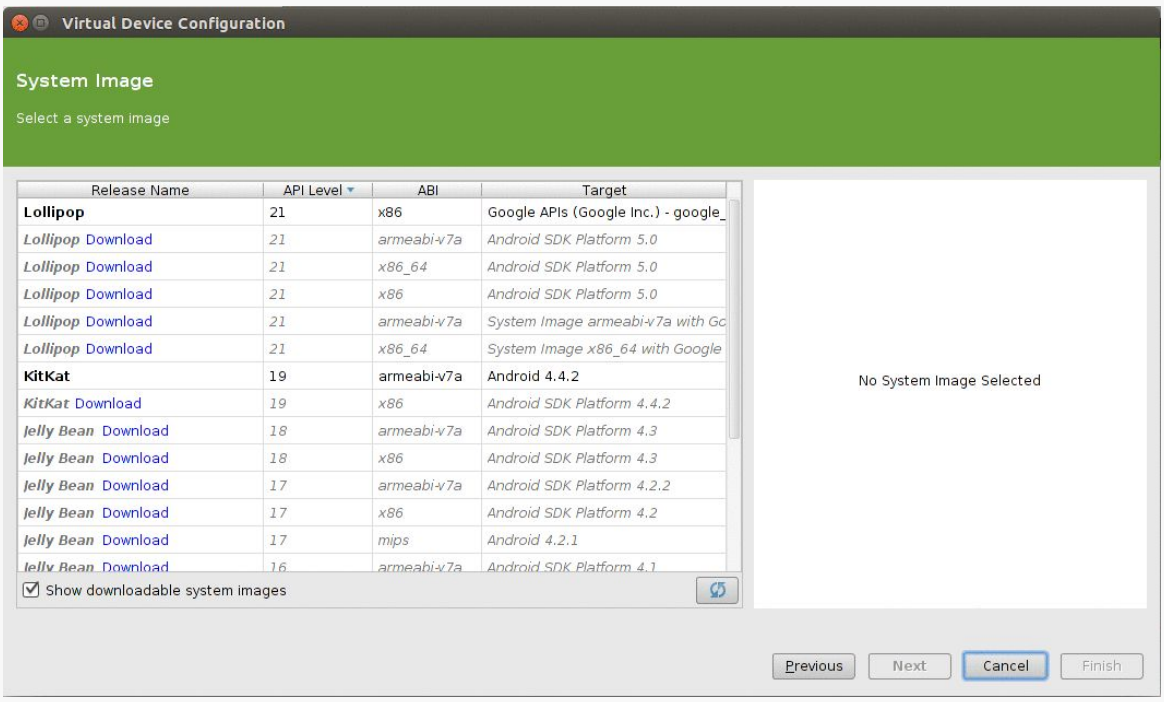


Figure 37: Android Studio Virtual Device Configuration Wizard, Second Page

For the tutorials in this book, you want an API Level 18 or 21 image, and for the `armeabi-v7` CPU architecture (unless you have gone ahead and configured x86 emulator support). You do not need one with the "Google APIs" — those are for emulators that have Google Play Services in them and related apps like Google Maps. The ones with "Download" next to them will trigger a one-time download of the files necessary to create AVDs for that particular API level and CPU architecture combination.

Clicking "Next" allows you to finalize the configuration of your AVD:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

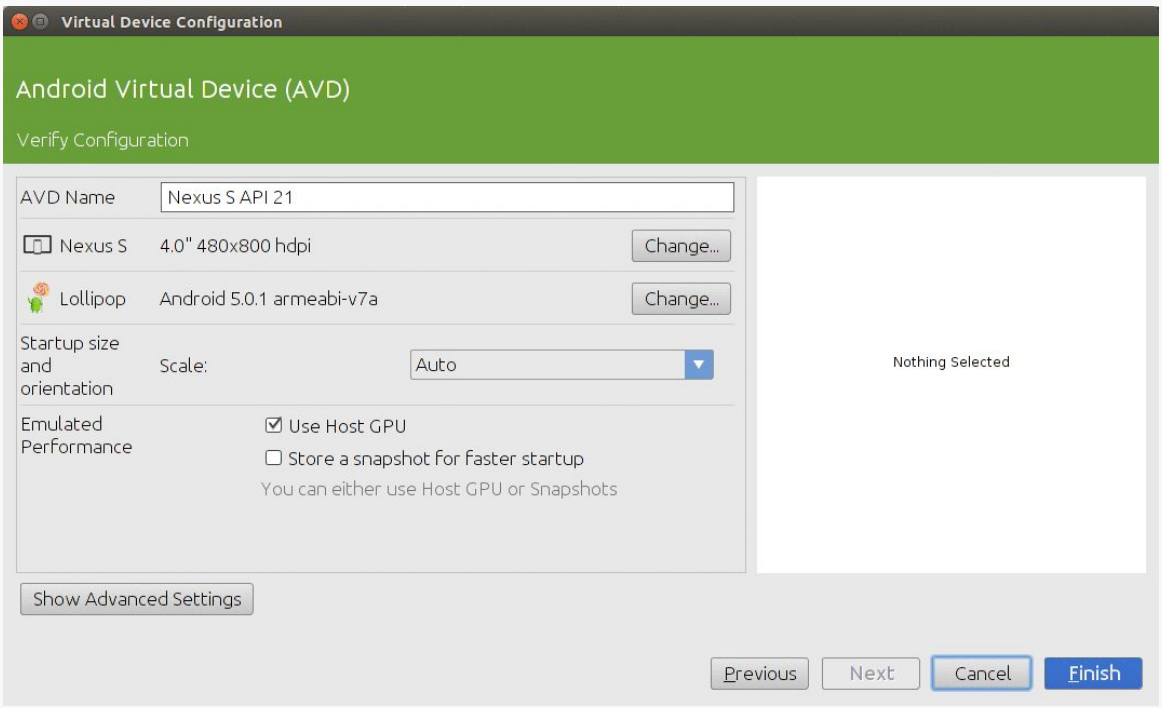


Figure 38: Android Studio Virtual Device Configuration Wizard, Third Page

A default name for the AVD is suggested, though you are welcome to replace this with your own value. The rest of the default values should be fine.

Clicking "Finish" will return you to the main AVD Manager, showing your new AVD. You can click "OK" to return to the main IDE window.

AVD Manager in Eclipse

To open the AVD Manager in Eclipse, choose Window > Android Virtual Device Manager from the main menu.

The AVD Manager starts up on a screen listing the AVDs you have available – initially, the list will be empty:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

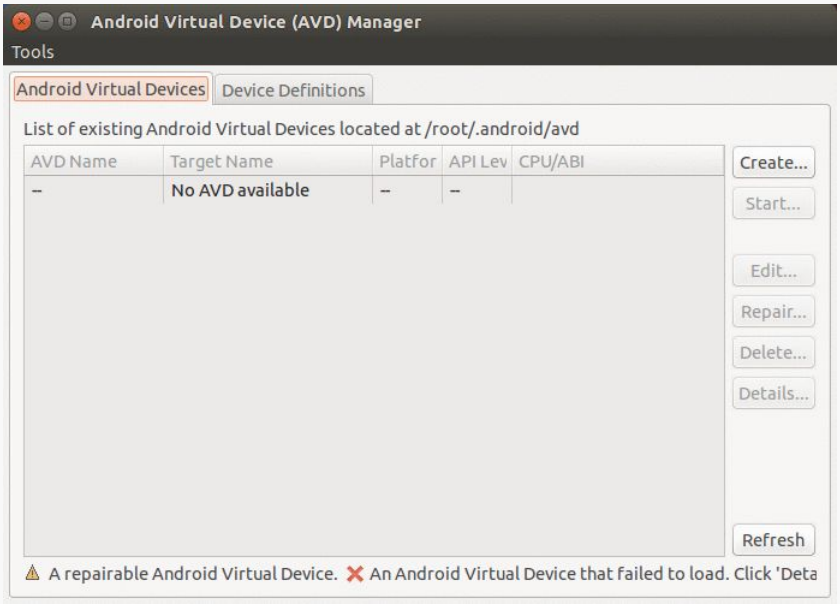


Figure 39: AVD Manager, From Fresh Installation

You will notice that there is a "Device Definitions" tab. This provides a catalog of device hardware configurations that you can use as the starting point for your emulator:

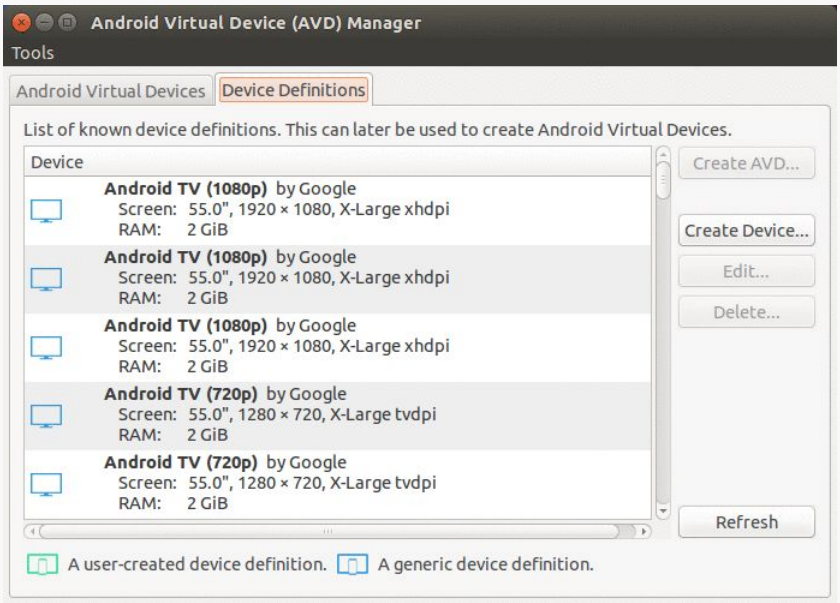


Figure 40: AVD Manager, Device Definitions Tab

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For now, though, on the "Android Virtual Devices" tab, click the New... button to create a new AVD file. This brings up a dialog where you can configure what this AVD should look and work like:

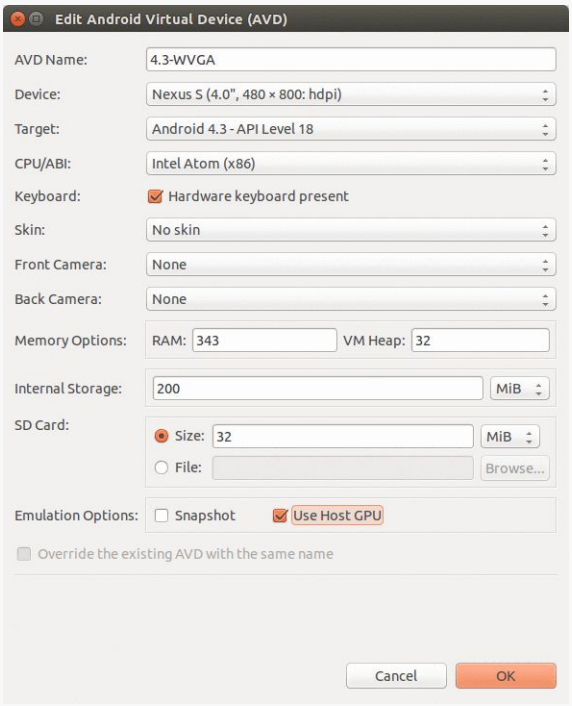


Figure 41: Adding a New AVD

You need to provide the following:

1. A name for the AVD. Since the name goes into files on your development machine, you will be limited by filename conventions for your operating system (e.g., no backslashes on Windows).
2. Which one of the available device templates from the "Device Definitions" tab you wish to use. Since the emulator runs slower with higher resolution screens, the Nexus S is a likely candidate, as it offers a not-too-unrealistic resolution and better emulator speed.
3. The Android version you want the emulator to run (a.k.a., the "target"). Choose one of the SDKs you installed via the drop-down list. Note that in addition to "pure" Android environments, you will have options based on the third-party add-ons you selected. For example, you probably have some options for setting up AVDs containing the Google APIs, and you will need such an AVD for testing an application that uses Google Maps.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

4. The CPU architecture your emulator will emulate. If you went ahead and set up x86 emulator support, choose your x86 emulator image here. Otherwise, choose your ARM emulator image.
5. Whether or not a hardware keyboard is present. Having this checked can ease your data entry on the emulator, as your development machine's keyboard will act as a keyboard for the emulated device.
6. Whether there should be a portion of the emulator window set aside to show hardware controls, such as a D-pad. This is usually a good idea, particularly while you are getting familiar with the Android environment.
7. Values for the memory and internal storage — the defaults are perfectly fine selections.
8. Details about the SD card the emulator should emulate. Since Android devices invariably have some form of "external storage", you probably want to set up an SD card, by supplying a size in the associated field. However, since a file will be created on your development machine of whatever size you specify for the card, you probably do not want to create a 2GB emulated SD card. 32MB is a nice starting point, though you can go larger if needed.
9. Whether or not "snapshot" mode is enabled. This can speed up restarting the emulator at the cost of hard disk space. For now, leave it unchecked.
10. Whether or not you wish to use the development machine's graphics card (GPU) to accelerate the emulator's graphics. Usually, this helps emulator performance, so checking that is worth trying. If you encounter problems running the emulator, try editing the AVD definition and unchecking this value.

Click the OK button, and your AVD stub will be created.

To start the emulator, highlight it in the list and click "Start...". You can skip the launch options for now and just click Launch. The first time you launch a new AVD, it will take a long time to start up. The second and subsequent times you start the AVD, it will come up a bit faster, and usually you only need to start it up once per day (e.g., when you start development). You do not need to stop and restart the emulator every time you want to test your application, in most cases. Also, Eclipse will automatically start an emulator if you do not have one started and you try running an application.

The emulator will go through a few startup phases, typically first with a plain-text "ANDROID" label (for pre-Android 4.0) or a blank screen (for Android 4.0+):

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 42: Android emulator, initial startup segment

... then a graphical Android logo:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 43: Android emulator, secondary startup segment

before eventually landing at the home screen, a welcome page (shown below, for Android 4.0), or the keyguard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

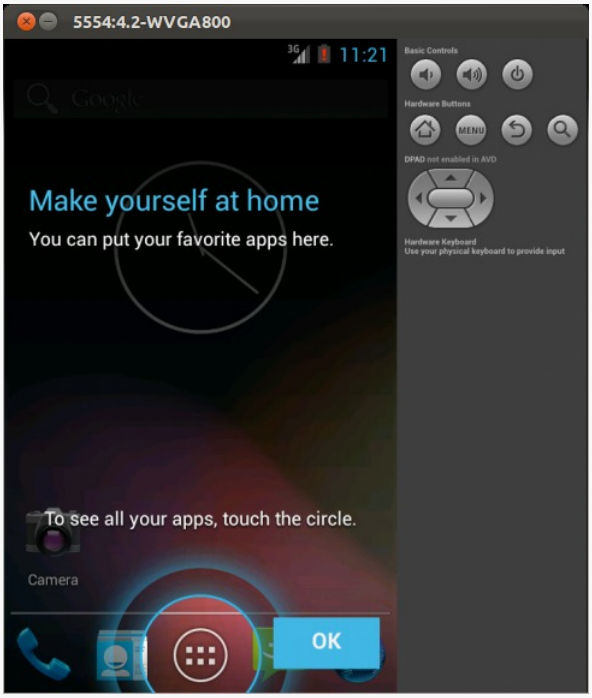


Figure 44: Android 4.0 emulator welcome page

If you get the keyguard (shown below), press the MENU button, or slide the lock on the screen to the right, to get to the emulator's home screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

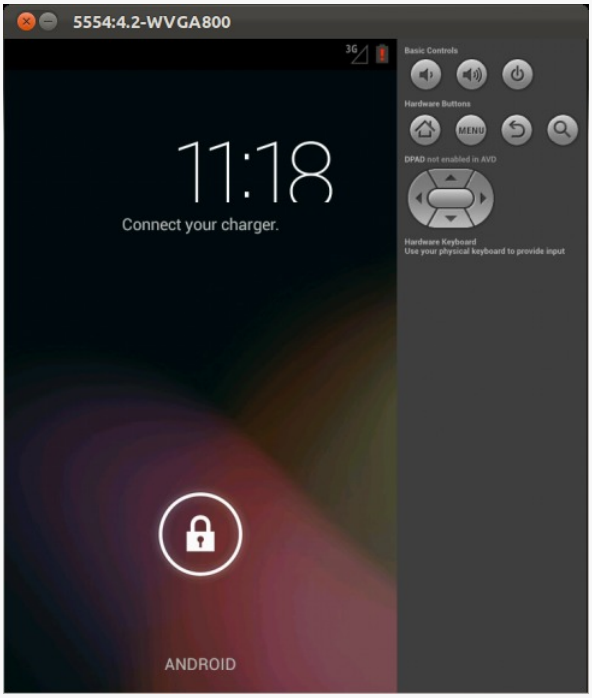


Figure 45: Android 4.0 keyguard

Step #3 - Set Up the Device

You do not need an Android device to get started in Android application development. Having one is a good idea before you try to ship an application (e.g., upload it to the Play Store). And, perhaps you already have a device – maybe that is what is spurring your interest in developing for Android.

If you do not have an Android device that you wish to set up for development, skip this step.

The first step to make your device ready for use with development is to go into the Settings application on the device. What happens now depends a bit on your Android version:

- On Android 1.x/2.x, go into Applications, then into Development
- On Android 3.0 through 4.1, go into "Developer options" from the main Settings screen

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- On Android 4.2 and higher, go into About, tap on the build number seven times, then press BACK, and go into "Developer options" (which was formerly hidden)

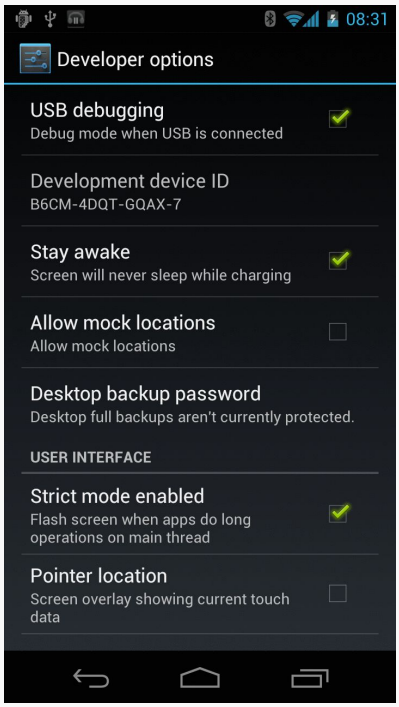


Figure 46: Android device development settings

You may need to slide a switch in the upper-right corner of the screen to the "ON" position to modify the values on this screen.

Generally, you will want to enable USB debugging, so you can use your device with the Android build tools. You can leave the other settings alone for now if you wish, though you may find the "Stay awake" option to be handy, as it saves you from having to unlock your phone all of the time while it is plugged into USB.

Note that on Android 4.2.2 and higher devices, before you can actually use the setting you just toggled, you will be prompted to allow USB debugging with your *specific* development machine via a dialog box:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

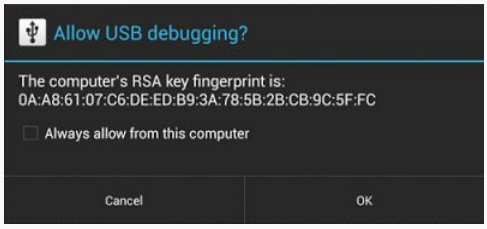


Figure 47: Allow USB Debugging Dialog

This occurs when you plug in the device via the USB cable and have the driver appropriately set up. That process varies by the operating system of your development machine, as is covered in the following sections.

Windows

When you first plug in your Android device, Windows will attempt to find a driver for it. It is possible that, by virtue of other software you have installed, that the driver is ready for use. If it finds a driver, you are probably ready to go.

If the driver is not found, here are some options for getting one.

Windows Update

Some versions of Windows (e.g., Vista) will prompt you to search Windows Update for drivers. This is certainly worth a shot, though not every device will have supplied its driver to Microsoft.

Standard Android Driver

In your Android SDK installation, if you chose to install the "Google USB Driver" package from the SDK Manager, you will find an `extras/google/usb_driver/` directory, containing a generic Windows driver for Android devices. You can try pointing the driver wizard at this directory to see if it thinks this driver is suitable for your device. This will often work for Nexus devices.

Manufacturer-Supplied Driver

If you still do not have a driver, the OEM USB Drivers in the developer documentation may help you find one for download from your device manufacturer. Note that you may need the model number for your device, instead of the model

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

name used for marketing purposes (e.g., GT-P3113 instead of "Samsung Galaxy Tab 2 7.0").

OS X and Linux

Odds are decent that simply plugging in your device will "just work". You can see if Android recognizes your device via running `adb devices` in a shell (e.g., OS X Terminal), where `adb` is in your `platform-tools/` directory of your SDK. If you get output similar to the following, the build tools detected your device:

```
List of devices attached
HT9CPP809576  device
```

If you are running Ubuntu (or perhaps other Linux variants), and this command did not work, you may need to add some `udev` rules. For example, here is a `51-android.rules` file that will handle the devices from a handful of manufacturers:

```
SUBSYSTEM=="usb", SYSFS{idVendor}=="0bb4", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="22b8", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="18d1", MODE="0666"
SUBSYSTEMS=="usb", ATTRS{idVendor}=="18d1", ATTRS{idProduct}=="0c01",
MODE="0666", OWNER="[me]"
SUBSYSTEM=="usb", SYSFS{idVendor}=="19d2", SYSFS{idProduct}=="1354", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="04e8", SYSFS{idProduct}=="681c", MODE="0666"
```

Drop that in your `/etc/udev/rules.d` directory on Ubuntu, then either reboot the computer or otherwise reload the `udev` rules (e.g., `sudo service udev reload`). Then, unplug and re-plug in the device and see if it is detected.

The CyanogenMod project maintains a page on their wiki with more on these `udev` rules, including rules from a variety of manufacturers and devices.

Step #4: Running the Project

Now, we can confirm that our project is set up properly by running it on a device or emulator. Once again, there are separate sections of instructions below for Android Studio versus Eclipse development — please follow the instructions that are appropriate for you.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio

Press the Run toolbar button (usually depicted as a green rightward-pointing triangle).

You will then be presented with a dialog indicating where you want the app to run: on some existing device or emulator, or on some newly-launched emulator:

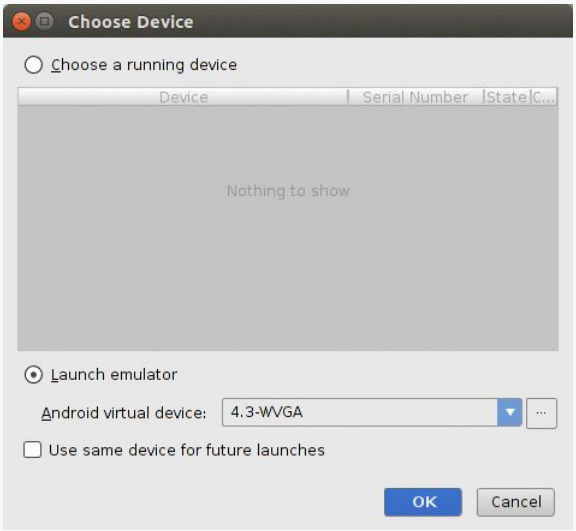


Figure 48: Android Studio Device Chooser Dialog

If you do not have an emulator running, choose one from the drop-down menu towards the bottom of the dialog, then click OK. Android Studio will launch your emulator for you.

And, whether you start a new emulator instance or reuse an existing one, your app should appear on it.

Eclipse

Press the Run toolbar button (usually depicted as a white "play" triangle in a green circle). The first time you run the project, you will see a "Run As" dialog, prompting you to declare how you want to run the app:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 49: Eclipse Run As Dialog

Click on "Android Application" and click "OK" to proceed.

At this point, if you have a compatible running emulator or device, the app will be installed and run on it. If you have more than one, you will be given a dialog where you can choose what to use. Otherwise, Eclipse will start up a suitable emulator, from the AVDs you created previously, then will install and run the app on it.

Results

You should see your app on the device or emulator:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

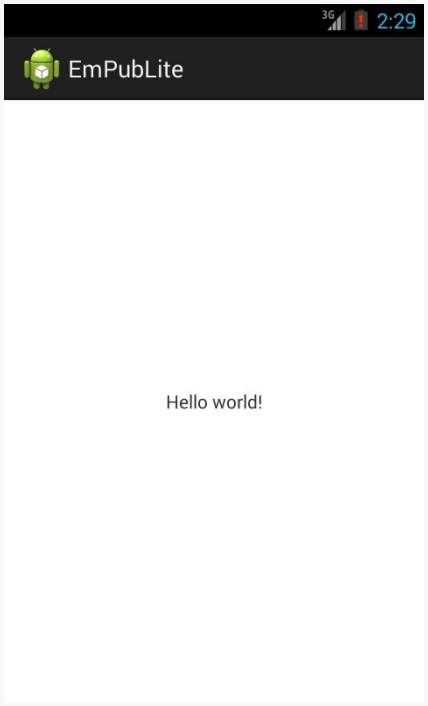


Figure 50: Android 4.3 Emulator with EmPubLite

Note that you will have to unlock your device or emulator to actually see the app running — it will not unlock automatically for you.

In Our Next Episode…

… we will modify the `AndroidManifest.xml` file of our tutorial project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Getting Around Android Studio

Eclipse has been around for a very long time and has proven to be a very popular IDE. As a result, there is quite a bit of material written about it, from books and blogs to Stack Overflow questions and official project documentation.

Android Studio shares a lot of functionality with its parent, IntelliJ IDEA. However, IDEA itself has not achieved Eclipse's level of popularity, even though it has long been the IDE of choice for many "power developers". And Android Studio's changes to IDEA are largely undocumented.

Hence, this chapter will serve as a quick tour of the Android Studio IDE, to help you get settled in. Other Android-specific capabilities of Android Studio will be explored in the chapters that follow.

Navigating The Project Explorer

After the main editing area — where you will modify your Java source code, your resources, and so forth — the piece of Android Studio you will spend the most time with is the project explorer, usually available on the left side of the IDE window:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

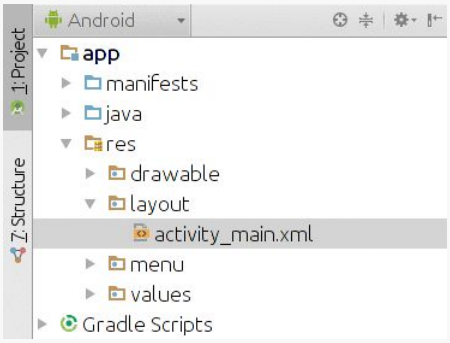


Figure 51: Android Studio Project Explorer, Showing Android Project View

This explorer pane has two main "project views" that an Android developer will use: the Android project view and the classic project view.

Android Project View

By default, when you create or import a project, you will wind up in the Android project view.

In theory, the Android project view is designed to simplify working with Android project files. In practice, it may do so, but only for some advanced developers. On the whole, it makes the IDE significantly more complicated for newcomers to Android, as it is rather difficult to see where things are and what relates to what.

We will return to the Android project view a bit later in the book and explain its benefits relative to resources and Gradle's sourcesets.

However, for most of the book — most importantly, for the tutorials – we will use the classic project view.

Classic Project View

To switch to the classic project view, click the drop-down just above the tree in the explorer, and choose Project:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

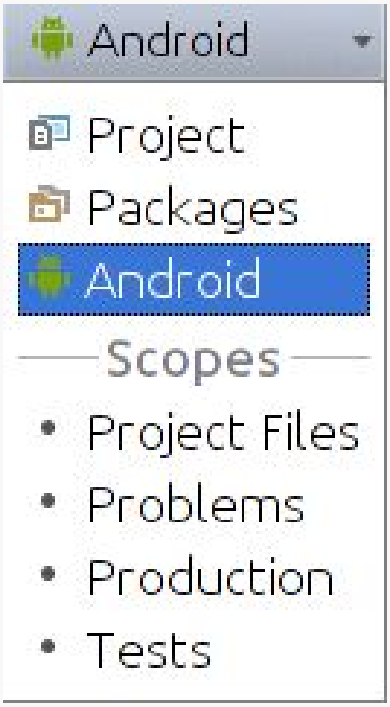


Figure 52: Android Studio Project Explorer, Showing Project View Drop-Down

This will change the contents of the tree to show you all of the files, in their associated directories:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

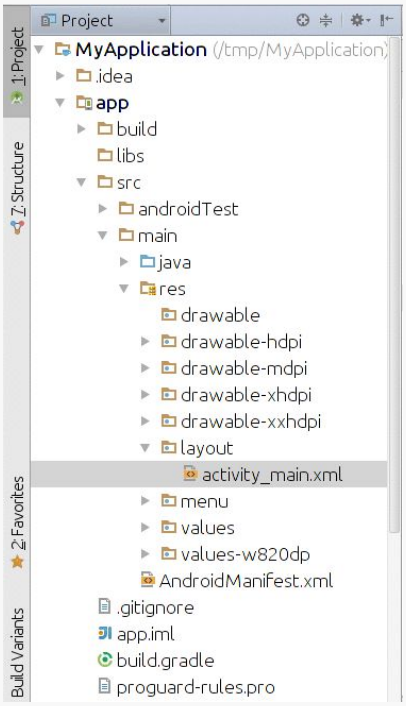


Figure 53: Android Studio Project Explorer, Showing Classic Project View

This project view is much like its equivalent in other IDEs, allowing you to find all of the pieces of your Android project. We will be exploring what those pieces are, and how their files are organized in our projects, in the next chapter.

Context Menus in the Explorer

Right-clicking over a directory or file in the explorer will give you a context menu with a variety of options. Some of these will be typical of any sort of file manager, such as "cut", "copy", and/or "paste" options. Some of these will be organized according to how IntelliJ IDEA manages application development, such as the "Refactor" sub-menu, where you can rename or move files around. Yet others will be specific to Android Studio, such as the ability to invoke wizards to create certain types of Android components or other Java classes.

Running Projects

As noted in Tutorial #2, to run your project, just press the Run toolbar button (usually depicted as a green rightward-pointing triangle). You will then be presented

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

with a dialog indicating where you want the app to run: on some existing device or emulator, or on some newly-launched emulator:

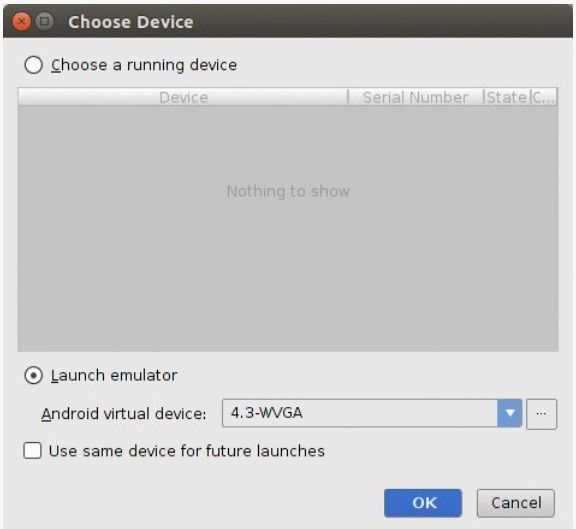


Figure 54: Android Studio Device Chooser Dialog

If you do not have an emulator running, choose one from the drop-down menu towards the bottom of the dialog, then click OK. Android Studio will launch your emulator for you. Whether you start a new emulator instance, reuse an existing one, or request that your app run on an attached Android device, your app should appear on it.

Viewing Output

Beyond your app itself, Android Studio will generate other sorts of diagnostic output, in the form of "console"-style transcripts of things that have occurred. The two of these that probably will matter most for you are the Gradle console and LogCat.

Gradle Console

By default, docked in the lower-right corner of your Android Studio window is a "Gradle Console" item. Tapping on that will open up a pane showing the output of attempts to build your application:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

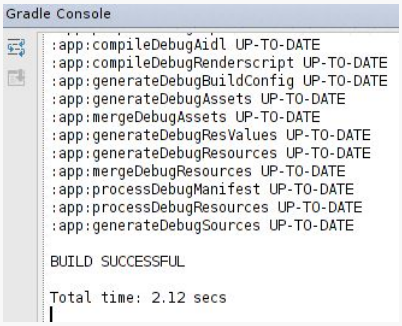


Figure 55: Android Studio Gradle Console

This may automatically appear from time to time, if specific build problems are detected, and you can always go examine it whenever you need.

Click on the "Gradle Console" item again to collapse the view and get it out of your way.

LogCat

Messages that appear at runtime — including the all-important Java stack traces triggered by bugs in your code — are visible in LogCat. The "Android" item docked towards the lower-left corner of your Android Studio window will display LogCat when tapped:

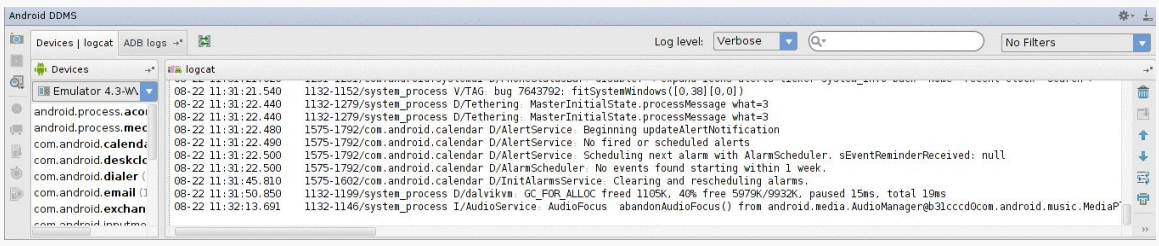


Figure 56: Android Studio LogCat View

LogCat is explained in greater detail a bit later in this book.

Accessing Android Tools

Not everything related to Android is directly part of Android Studio itself. In some cases, tools need to be shared between users of Android Studio, users of Eclipse, and users of "none of the above". In some cases, while the long term direction may be to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

incorporate the tools' functionality directly into Android Studio, that work simply has not been completed to date.

Here are some noteworthy Android-related tools that you can access via the Tools > Android main menu option.

SDK and AVD Managers

As we saw in Tutorial #1, the SDK Manager is Android's tool for downloading pieces of the Android SDK, including:

- "SDK Platform" editions, allowing us to compile against a particular API level
- ARM and (sometimes) x86 emulator images
- Documentation
- Updates to the core build tools
- Etc.

You can launch the SDK Manager via Tools > Android > SDK Manager from the Android Studio main menu, or by clicking on the "droid in a box" toolbar button:

Figure 57: Android Studio SDK Manager Toolbar Icon

The AVD Manager is the tool for creating emulators that emulate certain Android environments, based upon API level, screen size, and other characteristics.

You can launch the AVD Manager via Tools > Android > AVD Manager from the Android Studio main menu, or by clicking the "droid and a screen" toolbar button:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 58: Android Studio AVD Manager Toolbar Icon

Android Device Monitor

Elsewhere in this book, you will see references to tools associated with the Dalvik Device Monitor Service (DDMS), such as using it to help inspect your running apps for memory or threading issues. You will also see references to tools like Hierarchy View, for trying to make sense of your UI as it appears at runtime, after you have programmatically made lots of changes to it.

In Eclipse, DDMS and Hierarchy View are "perspectives", added to Eclipse via the ADT plugin.

For everyone not using Eclipse — including Android Studio users — DDMS and Hierarchy View are available via the Android Device Monitor standalone tool. Android Studio users can launch the Monitor via Tools > Android > Android Device Monitor from the main menu.

This will first bring up a splash screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 59: Android Device Monitor Splash Screen

followed by the Monitor itself:

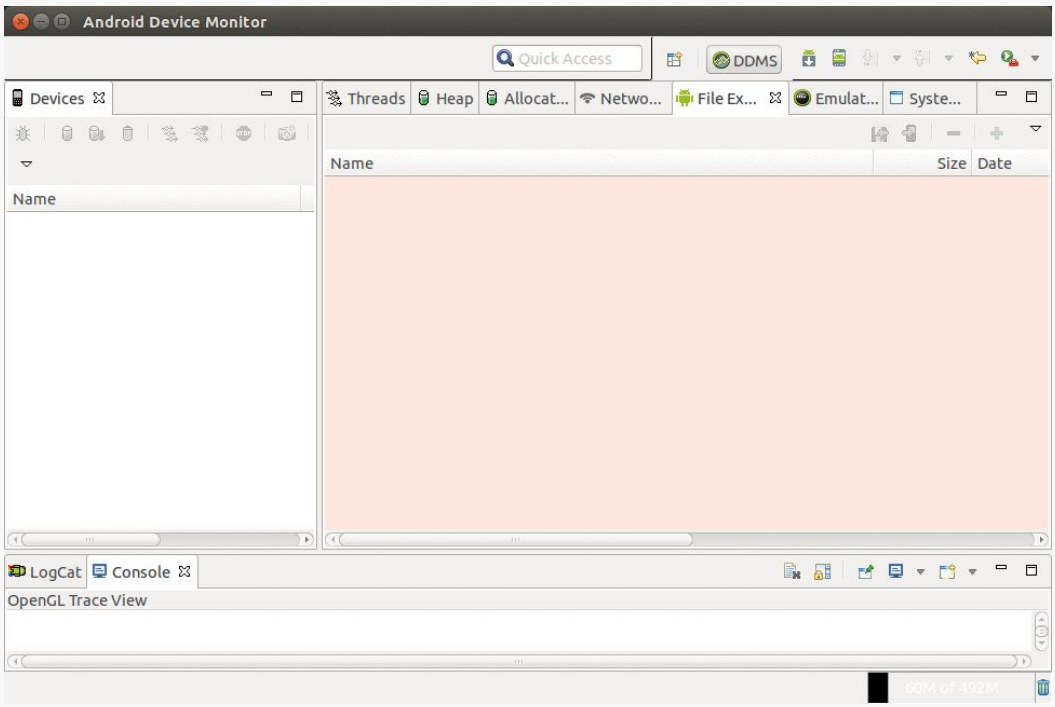


Figure 60: Android Device Monitor, As Initially Opened

If you read about things available from DDMS or Hierarchy View online, such as in blog posts or Stack Overflow answers, most of those capabilities should be available to you via the Android Device Monitor.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio and Release Channels

When you install Android Studio for the first time, your installation will be set up to get updates on the "stable" release channel. Here, a "release channel" is a specific set of possible upgrades. The "stable" release channel means that you are getting full production-ready updates. Android Studio will check for updates when launched, and you can manually check for updates via Help > Check for Update... in the main menu.

If an update is available, you will be presented with a dialog box showing you details of the update:

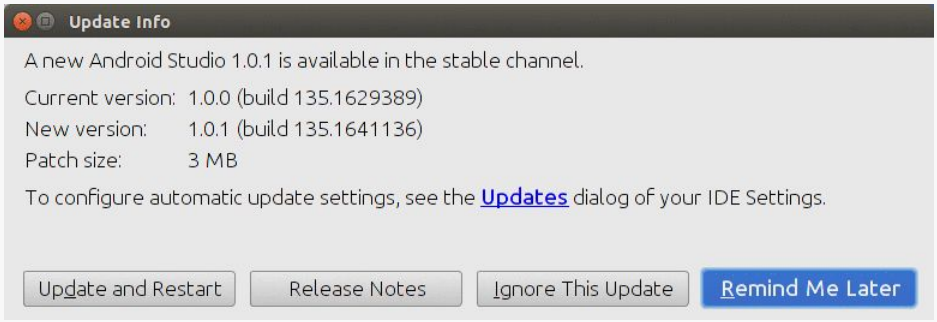


Figure 61: Android Studio Update Dialog

Choosing "Release Notes" will bring up a Web page with release notes for the new release. Clicking "Update and Restart" does pretty much what the button name suggests: it downloads the update and restarts the IDE, applying the update along the way.

Clicking the "Updates" hyperlink in the dialog brings up yet another dialog, allowing you to choose which release channel you want to subscribe to:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

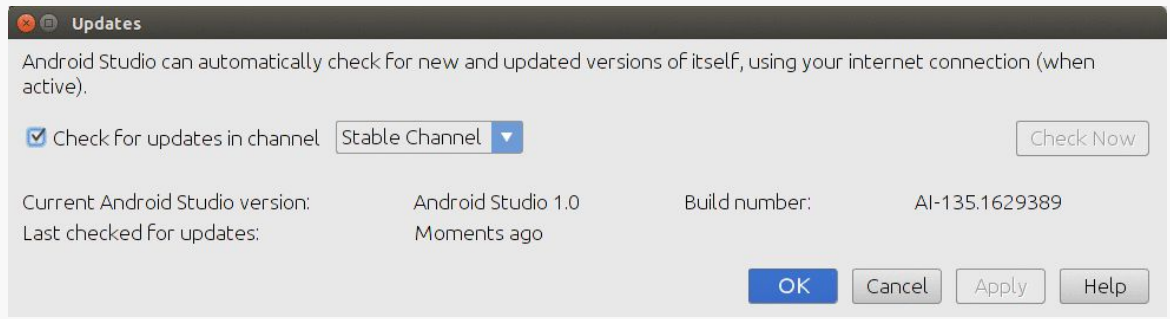


Figure 62: Android Studio Update Release Channel Dialog

You have four channels to choose from:

- Stable, which is appropriate for most developers
- Beta, which will get updates that are slightly ahead of stable
- Dev, which is even more ahead than is the beta channel
- Canary, which is updated *very* early (and the name, suggestive of a "canary in a coal mine", indicates that you are here to help debug the IDE)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Contents of Android Projects

The Android build system is organized around a specific directory tree structure for your Android project, much like any other Java project. The specifics, though, are fairly unique to Android — the Android build tools do a few extra things to prepare the actual application that will run on the device or emulator.

Making things more complicated is that the default structure is different for the classic tools (e.g., Eclipse) and the new generation of tools (e.g., Android Studio).

Here is a quick primer on the project structure, to help you make sense of it all, particularly for the sample code referenced in this book.

What You Get, In General

The details of *exactly* what files are in your project depend upon your choice of IDE. However, regardless of whether you go with Android Studio or Eclipse, there are many elements in common.

The Manifest

`AndroidManifest.xml` is an XML file describing the application being built and what components — activities, services, etc. — are being supplied by that application. You can think of it as being the "table of contents" of what your application is about, much as a book has a "table of contents" listing the various parts, chapters, and appendices that appear in the book.

We will examine the manifest a bit more closely starting in the next chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Java

When you created the project, you supplied the fully-qualified class name of the "main" activity for the application (e.g., `com.commonsware.android.SomeDemo`). You will then find that your project's Java source tree already has the package's directory tree in place, plus a stub `Activity` subclass representing your main activity (e.g., `src/com/commonsware/android/SomeDemoActivity.java`). You are welcome to modify this file and add Java classes as needed to implement your application, and we will demonstrate that countless times as we progress through this book.

Elsewhere — in directories that you normally do not work with — the Android build tools will also be code-generating some source code for you each time you build your app. One of the code-generated Java classes (`R.java`) will be important for controlling our user interfaces from our own Java code, and we will see many references to this `R` class as we start building applications in earnest.

The Resources

You will also find that your project has a `res/` directory tree. This holds "resources" — static files that are packaged along with your application, either in their original form or, occasionally, in a preprocessed form. Some of the subdirectories you will find or create under `res/` include:

1. `res/drawable/` for images (PNG, JPEG, etc.)
2. `res/layout/` for XML-based UI layout specifications
3. `res/menu/` for XML-based menu specifications
4. `res/raw/` for general-purpose files (e.g., an audio clip, a CSV file of account information)
5. `res/values/` for strings, dimensions, and the like
6. `res/xml/` for other general-purpose XML files you wish to ship

Some of the directory names may have suffixes, like `res/drawable-hdpi/`. This indicates that the directory of resources should only be used in certain circumstances — in this case, the drawable resources should only be used on devices with high-density screens.

We will cover all of these, and more, later in this book.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Build Instructions

The IDE needs to know how to take all of this stuff and come up with an Android APK file. Some of this is already "known" to the IDE based upon how the IDE was written. But some details are things that you may need to configure from time to time, and so those details are stored in files that you will edit, in one means or another, from your IDE.

In Android Studio, most of this knowledge is kept in one or more files named `build.gradle`. These are for a build engine known as Gradle, that Android Studio uses to build APKs and other Android outputs.

In Eclipse, this knowledge is scattered among several files, some of which you might edit manually (e.g., `project.properties`) and some of which you would only change through Eclipse itself (e.g., `.classpath`).

The Contents of an Android Studio Project

An Android Studio project is significantly more complex than is an Eclipse project. That complexity is designed to give you more power when you become an expert Android developer.

In the short term, though, it may be a bit confusing.

The Root Directory

In the root directory of your project, the most important item is the `app/` directory, where your application code resides. We will look at that in the next section.

Beyond the `app/` directory, the other noteworthy files in the root of your project include:

- `build.gradle`, which is part of the build instructions for your project, as is described above
- Various other Gradle-related files (`settings.gradle`, `gradle.properties`, and so forth)
- `local.properties`, which indicates where your Android SDK tools reside
- An `.iml` file, where Android Studio holds some additional metadata about your project

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The App Directory

The `app/` directory, and its contents, are where you will spend most of your time as a developer. Rarely do you need to manipulate the files in the project root.

The most important thing in the `app/` directory is the `src/` directory, which is the root of your project's sourcesets, which will be described in the next section.

Beyond the `src/` directory, there are a few other items of note in `app/`:

- A `build/` directory, which will hold the outputs of building your app, including your APK file
- A `build.gradle` file, where most of your project-specific Gradle configuration will go, to teach Android Studio how to build your app
- An `app.iml` file, containing more Android Studio metadata

The Sourcesets

Sourcesets are where the "source" of your project is organized. Here, "source" not only refers to programming language source code (e.g., Java), but other types of inputs to the build, such as your resources.

The sourceset that you will spend most of your time in is `main/`. You will also have a stub sourceset named `androidTest`, for use in creating unit tests, as will be covered later in the book.

Inside of a sourceset, you can have:

- Java code, in a `java/` directory
- Resources, in a `res/` directory
- Assets, in an `assets/` directory, representing other static files you wish packaged with the application for deployment onto the device
- Your `AndroidManifest.xml` file

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

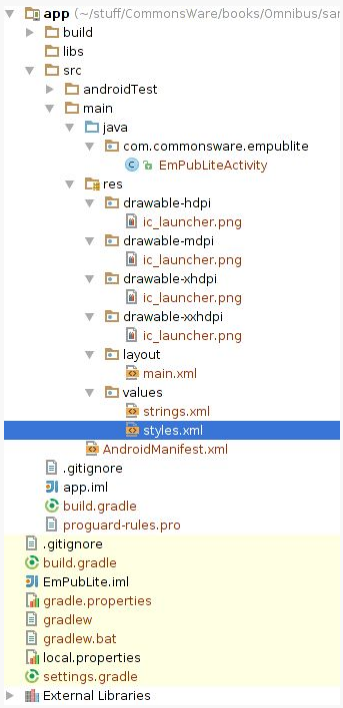


Figure 63: Android Studio Project Explorer, Showing EmPubLite

The Contents of an Eclipse Project

When you create a new Android project in Eclipse, you get several items in the project's root directory, including:

1. `AndroidManifest.xml`, as is described above
2. `bin/`, which holds the application once it is compiled (note: this directory will be created when you first build your application)
3. `res/`, which holds your resources, as is described above
4. `src/`, which holds the Java source code for the application

In addition to the files and directories shown above, you may find any of the following in Android projects:

1. `assets/`, which holds other static files you wish packaged with the application for deployment onto the device
2. `gen/`, where Android's build tools will place source code that they generate
3. `libs/`, which holds any third-party Java JARs your application requires
4. `*.properties`, containing configuration data for your builds

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

5. `proguard.cfg` or `proguard-project.txt`, which are used for integration with ProGuard for obfuscating your Android code
6. Hidden Eclipse project files (e.g., `.classpath`)

What You Get Out Of It

As part of running your app on a device or emulator, the IDE will generate an APK file. You will find this:

- in the `build/outputs/apk` directory of your Android Studio project, if the project has no modules (e.g., no `app/` directory), or
- in the `build/outputs/apk` directory of your module's directory, (e.g., `app/build/outputs/apk` for a traditional Android Studio project), or
- in the `bin/` directory of your Eclipse project

The APK file is a ZIP archive containing your compiled Java classes, the compiled edition of your resources (`resources.arsc`), any un-compiled resources (such as what you put in `res/raw/`), and the `AndroidManifest.xml` file. If you build a debug version of the application — which is the default — you will have `yourapp-debug.apk` as your APK, for an app named `yourapp`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Introducing Gradle and the Manifest

In the discussion of Android Studio, this book has mentioned something called "Gradle", without a lot of explanation.

In this chapter, the mysteries of Gradle will be revealed to you.

(well, OK, *some* of the mysteries...)

We also mentioned in passing in the previous chapter the concept of the "manifest", as being a special file in our Android projects.

On the one hand, Gradle and the manifest are not strictly related. On the other hand, some (but far from all) of the things that we can set up in the manifest can be overridden in Gradle... for Android Studio users. Since Eclipse users are not using Gradle, their definitions will always be in the manifest itself.

So, in this chapter, we will review both what Gradle is, what the manifest is, what each of their roles are, and the basics of how they tie together.

Gradle: The Big Questions

First, let us "set the stage" by examining what this is all about, through a series of fictionally-asked questions (FAQs).

What is Gradle?

Gradle is software for building software, otherwise known as "build automation software" or "build systems". You may have used other build systems before in other environments, such as `make` (C/C++), `rake` (Ruby), Ant (Java), Maven (Java), etc.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

These tools know — via intrinsic capabilities and rules that you teach them — how to determine what needs to be created (e.g., based on file changes) and how to create them. A build system does not compile, link, package, etc. applications directly, but instead directs separate compilers, linkers, and packagers to do that work.

Gradle uses a domain-specific language (DSL) built on top of Groovy to accomplish these tasks.

What is Groovy?

There are many programming languages that are designed to run on top of the Java VM. Some of these, like JRuby and Jython, are implementations of other common programming languages (Ruby and Python, respectively). Other languages are unique, and Groovy is one of those.

Groovy scripts look a bit like a mashup of Java and Ruby. As with Java, Groovy supports:

- Defining classes with the `class` keyword
- Creating subclasses using `extends`
- Importing classes from external JARs using `import`
- Defining method bodies using braces (`{` and `}`)
- Objects are created via the `new` operator

As with Ruby, though:

- Statements can be part of a class, or simply written in an imperative style, like a scripting language
- Parameters and local variables are not typed
- Values can be automatically patched into strings, though using slightly different syntax (`"Hello, $name"` for Groovy instead of `"Hello, #{name}"` for Ruby)

Groovy is an interpreted language, like Ruby and unlike Java. Groovy scripts are run by executing a **`groovy`** command, passing it the script to run. The Groovy runtime, though, is a Java JAR and requires a JVM in order to operate.

One of Groovy's strengths is in creating a domain-specific language (or DSL). Gradle, for example, is a Groovy DSL for doing software builds. Gradle-specific capabilities appear to be first-class language constructs, generally indistinguishable

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

from capabilities intrinsic to Groovy. Yet, the Groovy DSL is largely declarative, like an XML file.

To some extent, we get the best of both worlds: XML-style definitions (generally with less punctuation), yet with the ability to "reach into Groovy" and do custom scripting as needed.

What Does Android Have To Do with Gradle?

Google has published the Gradle for Android plugin, which gives Gradle the ability to build Android projects. Google is also using Gradle and Gradle for Android as the build system behind Android Studio.

Why Did We Move to Gradle?

Originally, when we would build an app, those builds were done using Eclipse and Ant. Eclipse was the IDE, while Ant was the command-line tool. Eclipse does not use Ant for building Android projects, but rather has its own build system. And we were successfully building a million-plus apps using these tools. Those tools still work today, though Ant support is fading fast.

So, why change?

There appear to be several contributing factors, including:

- Maintaining two separate build systems (Ant and Eclipse's native approach) was becoming time-consuming, and would become worse with the advent of Android Studio and yet another build system. Hence, Google wishes to standardize on a single build system, based upon Gradle, for IDE and command-line scenarios.
- Getting Ant scripts to do everything that Google needed for builds was getting a bit creaky.
- Ant has no first-class support for "external artifacts" (e.g., libraries) and dependency management of those libraries. While there are ways to graft Maven onto Ant, or use Maven's own build system, Google never endorsed that approach. Gradle offers much better support in this area than do Eclipse or Ant, and will help make it easier for developers to reliably consume libraries from a variety of authors.
- Gradle is designed to be integrated into IDEs as a library, much more than Ant is.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

As to why Google chose Gradle over Maven... you would have to ask Google.

How Does Gradle Relate to Android Studio?

As noted above, Android Studio uses the new Gradle-based build system as its native approach for building Android projects. While the IntelliJ IDEA IDE that serves as Android Studio's core also has its own build system (much like Eclipse has one), IDEA is more amenable to replaceable build systems.

Over time, this will allow Google to focus on a single build system (Gradle) for all supported scenarios, rather than having to deal with a collection of independent build systems.

How Does Gradle Relate to Eclipse?

At the time of this writing, Eclipse has no ability to use Gradle for Android build scripts, either directly or as a source of project configuration data for imports. The Android Developer Tools (ADT) plugin for Eclipse does have the ability to export a generated Gradle build file for a project.

It is unclear to what extent that Eclipse will ever get Gradle for Android support. It is best to assume that this will not be possible any time soon.

Obtaining Gradle

As with any build system, to use it, you need the build system's engine itself.

If you will only be using Gradle in the context of Android Studio, the IDE will take care of getting Gradle for you. If, however, you are planning on using Gradle outside of Android Studio (e.g., command-line builds), you will want to consider where your Gradle is coming from. This is particularly important for situations where you want to build the app with no IDE in sight, such as using a continuous integration (CI) server, like Jenkins.

Direct Installation

What most developers looking to use Gradle outside of Android Studio will wind up doing is installing Gradle directly.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Gradle download page contains links to ZIP archives for Gradle itself: binaries, source code, or both.

You can unZIP this archive to your desired location on your development machine.

Linux Packages

You may be able to obtain Gradle via a package manager on Linux environments. For example, there is an Ubuntu PPA for Gradle.

The gradlew Wrapper

If you are starting from a project that somebody else has published, you may find a `gradlew` and `gradlew.bat` file in the project root, along with a `gradle/` directory. This represents the "Gradle Wrapper".

The Gradle Wrapper consists of three pieces:

- the batch file (`gradlew.bat`) or shell script (`gradlew`)
- the JAR file used by the batch file and shell script (in the `gradle/wrapper/` directory)
- the `gradle-wrapper.properties` file (also in the `gradle/wrapper/` directory)

Android Studio uses the `gradle-wrapper.properties` file to determine where to download Gradle from, for use in your project, from the `distributionUrl` property in that file:

```
#Wed Apr 10 15:27:10 PDT 2013
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-2.2.1-all.zip
```

When you create or import a project, or if you change the version of Gradle referenced in the properties file, Android Studio will download the Gradle pointed to by the `distributionUrl` property and install it to a `.gradle/` directory (note the leading `.`) in your project. That version of Gradle will be what Android Studio uses.

RULE #1: Only use a `distributionUrl` that you trust.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you are importing an Android project from a third party — such as the samples for this book — and they contain the `gradle/wrapper/gradle-wrapper.properties` file, examine it to see where the `distributionUrl` is pointing to. If it is loading from `services.gradle.org`, or from an internal enterprise server, it is probably trustworthy. If it is pointing to a URL located somewhere else, consider whether you really want to use that version of Gradle, considering that it may have been tampered with.

The batch file, shell script, and JAR file are there to support command-line builds. If you use `gradlew`, it will use a local copy of Gradle installed in `.gradle/` in the project. If there is no such copy of Gradle, `gradlew` will download Gradle from the `distributionUrl`, as does Android Studio. Note that Android Studio does not use `gradlew` for this role — that logic is built into Android Studio itself.

RULE #2: Only use a `gradlew` that you REALLY trust.

It is relatively easy to examine a `.properties` file to check a URL to see if it seems valid. Making sense of a batch file or shell script can be cumbersome. Decompiling a JAR file and making sense of it can be rather difficult. Yet, if you use `gradlew` that you obtained from somebody, that script and JAR are running on your development machine, as is the copy of Gradle that they install. If that code was tampered with, the malware has complete access to your development machine and anything that it can reach, such as servers within your organization.

Note that you do not have to use the Gradle Wrapper at all. If you would rather not worry about it, install a version of Gradle on your development machine yourself and remove the Gradle Wrapper files. You can use the `gradle` command to build your app (if your Gradle's `bin/` directory is in your `PATH`), and Android Studio will use your Gradle installation (if you teach it where to find it, such as via the `GRADLE_HOME` environment variable).

Versions of Gradle and Gradle for Android

The Gradle for Android plugin that we will use to give Gradle "super Android powers!" is updated periodically. Each update has its corresponding required version of Gradle.

The rules, according to Google, are:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Android Studio 1.x works with any Gradle for Android plugin version 1.x, where the values for "x" do not need to match.
- Gradle for Android 1.x should work with all Gradle 2.x versions

If you are using the Gradle Wrapper, you are using an installation of Gradle that is local to the project. So long as the version of Gradle in the project matches the version of Gradle for Android requested in the `build.gradle` file — as will be covered in the next chapter — you should be in fine shape.

If you are not using the Gradle Wrapper, you will need to decide when to take on a new Gradle for Android release and plan to update your Gradle installation and `build.gradle` files in tandem at that point.

Gradle Environment Variables

If you installed Gradle yourself, you will want to define a `GRADLE_HOME` environment variable, pointing to where you installed Gradle, and to add the `bin/` directory inside of Gradle to your `PATH` environment variable.

You may also consider setting up a `GRADLE_USER_HOME` environment variable, pointing to a directory in which Gradle can create a `.gradle` subdirectory, for per-user caches and related materials. By default, Gradle will use your standard home directory.

Examining the Gradle Files

An Android Studio project may have two `build.gradle` files, one at the project level and one at the "module" level (e.g., in the `app/` directory). That split is not required for single-module projects — those could just as easily have one `build.gradle` file for the module and nothing else.

Regardless of whether your `build.gradle` file is in one or two pieces, there are a few common elements that you will find, outlined in this section.

buildscript

The `buildscript` closure (i.e., code section wrapped in braces) in Gradle is where you configure the JARs and such that Gradle itself will use for interpreting the rest of

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the file. Hence, here you are not configuring your *project* so much as you are configuring *the build itself*.

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}
```

The `repositories` closure inside the `buildscript` closure indicates where dependencies can come from, typically in the form of Maven-style repositories. Here, `mavenCentral()` is a built-in method that sets up the repository information for Maven Central, where Google is now publishing its required libraries.

The `dependencies` closure indicates what is required to be able to run the rest of the build script. `classpath 'com.android.tools.build:gradle:1.0.0'` is not especially well-documented by the Gradle team. However the `'com.android.tools.build:gradle:1.0.0'` portion means:

- Find the `com.android.tools.build` group of artifacts in a repository
- Find the `gradle` artifact within that group
- Ensure that we have version `1.0.0` of the artifact

The first time you run your build, with the `buildscript` closure as shown above, Gradle will notice that you do not have this dependency. It will then visit Maven Central and find the `com.android.tools.build` group and the `gradle` artifact within that group, then download version 1.0.0.

Sometimes, the last segment of the version is replaced with a `+` sign (e.g., `1.0.+`). This tells Gradle to download the latest version, thereby automatically upgrading you to the latest patch-level (e.g., `1.0.3` at some point).

apply plugin

The `'com.android.tools.build:gradle:1.0.0` dependency contains an implementation of the Gradle for Android plugin, to teach Gradle Android-specific constructs, such as those found in the `android` closure covered later in this chapter. However, just downloading the dependency does not actually apply anything in it.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `apply plugin: 'com.android.application'` statement actually adds the Gradle for Android plugin to Gradle for this build. In particular, it indicates that we are building an Android application.

dependencies

The `dependencies` closure in the `buildscript` closure specifies dependencies for the build script. The `dependencies` closure in the root of the `build.gradle` file, by contrast, specifies dependencies for the *app*.

However, the definition of "dependency" is largely the same: it is some library or similar artifact that we need in addition to our project's source code.

We will get into the concept of these libraries later in the book.

android

The `android` closure contains all of the Android-specific configuration information. This closure is what the Android plugin enables.

But before we get into what is in this closure, we should "switch gears" and talk about the manifest file, as what goes in the `android` closure is related to what goes in the manifest file.

Introducing the Manifest

The foundation for any Android application is the manifest file: `AndroidManifest.xml`. This will be in your project root for Eclipse projects and in your `app` module's `src/main/` directory for classic Android Studio projects.

Here is where you declare what is inside your application — the activities, the services, and so on. You also indicate how these pieces attach themselves to the overall Android system; for example, you indicate which activity (or activities) should appear on the device's main menu (a.k.a., launcher).

When you create your application, you will get a starter manifest generated for you. For a simple application, offering a single activity and nothing else, the auto-generated manifest will probably work out fine, or perhaps require a few minor modifications. On the other end of the spectrum, the manifest file for the Android

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

API demo suite is over 1,000 lines long. Your production Android applications will probably fall somewhere in the middle.

As mentioned previously, some items can be defined in both the manifest and in a `build.gradle` file. The approach of putting that stuff in the manifest still works, and for Eclipse it is the only option, as Eclipse is not looking at a `build.gradle` file. For Android Studio users, you will probably use the Gradle file and not have those common elements be defined in the manifest.

Things In Common Between the Manifest and Gradle

There are a few key items that can be defined in the manifest and can be overridden in `build.gradle` statements. These items are fairly important to the development and operation of our Android apps as well.

Package Name and Application ID

The root of all manifest files is, not surprisingly, a `manifest` element:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.empublite">
```

Note the `android` namespace declaration. You will only use the namespace on many of the attributes, not the elements (e.g., `<manifest>`, not `<android:manifest>`).

The biggest piece of information you need to supply on the `<manifest>` element is the `package` attribute.

The `package` attribute will always need to be in the manifest, even for Android Studio projects. The `package` attribute will control where some source code is generated for us, notably some `R` and `BuildConfig` classes that we will encounter later in the book.

Since the `package` value is used for Java code generation, it has to be a valid Java package name. Java convention says that the package name should be based off of a reverse domain name (e.g., `com.commonsware.empublite`), where you own the domain in question. That way, it is unlikely that anyone else will accidentally collide with the same name.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `package` also serves as our app's default "application ID". This needs to be a unique identifier, such that:

- no two apps can be installed on the same device at the same time with the same application ID
- no two apps can be uploaded to the Play Store with the same application ID (and other distribution channels may have the same limitation)

By default, the application ID is the `package` value, but Android Studio users can override it in their Gradle build files. Specifically, inside of the `android` closure can be a `defaultConfig` closure, and inside of there can be an `applicationId` statement:

```
android {
    // other stuff

    defaultConfig {
        applicationId "com.commonsware.empublite"
        // more other stuff
    }
}
```

Not only can Android Studio users override the application ID in the `defaultConfig` closure, but there are ways of having different application ID values for different scenarios, such as a debug build versus a release build. We will explore that more later in the book.

minSdkVersion and targetSdkVersion

Your manifest may also contain a `<uses-sdk>` element as a child of the `<manifest>` element, to specify what versions of Android you are supporting. It can contain, among other things, `android:minSdkVersion` and `android:targetSdkVersion` attributes. Eclipse projects will always have this element. Android Studio projects may not have this element, because the values are defined as `minSdkVersion` and `targetSdkVersion` properties in the `defaultConfig` closure, where `applicationId` can be defined.

Of the two, the more critical one is `minSdkVersion`. This indicates what is the oldest version of Android you are testing with your application. The value of the attribute is an integer representing the Android API level. So, if you are only testing your application on Android 4.1 and newer versions of Android, you would set your `minSdkVersion` to be `16`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You can also specify a `targetSdkVersion`. This indicates what version of Android you are thinking of as you are writing your code. If your application is run on a newer version of Android, Android may do some things to try to improve compatibility of your code with respect to changes made in the newer Android. Nowadays, most Android developers should specify a target SDK version of 14 or higher. We will start to explore more about the `targetSdkVersion` as we get deeper into the book; for the moment, whatever your IDE gives you as a default value is probably a fine starting point.

The XML element looks like:

```
<uses-sdk android:minSdkVersion="14" android:targetSdkVersion="19" />
```

The corresponding entries in `build.gradle` go in the `defaultConfig` closure:

```
android {
    // other stuff

    defaultConfig {
        applicationId "com.commonsware.empublite"
        minSdkVersion 15
        targetSdkVersion 18
        // more other stuff
    }
}
```

Version Code and Version Name

Your manifest can also specify `android:versionName` and `android:versionCode` attributes, up on the root `<manifest>` element. An Android Studio project, though, frequently skips those and defines them via `versionName` and `versionCode` properties in the `defaultConfig` closure.

These two values represent the versions of your application. The `versionName` value is what the user will see for a version indicator in the Applications details screen for your app in their Settings application:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

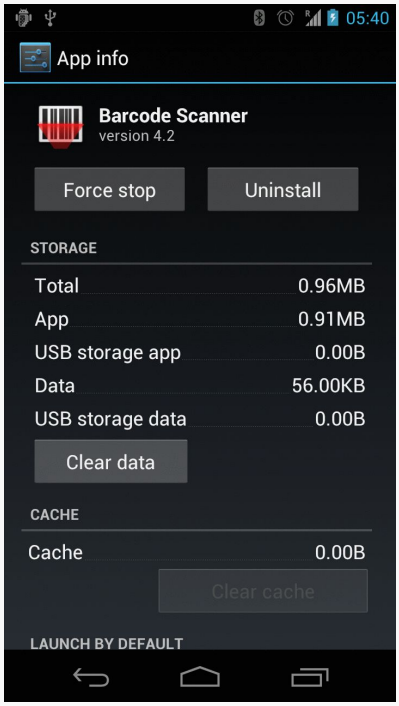


Figure 64: Barcode Scanner App Screen in Settings, Showing Version 4.2

Also, the version name is used by the Play Store listing, if you are distributing your application that way. The version name can be any string value you want.

The `versionCode`, on the other hand, must be an integer, and newer versions must have higher version codes than do older versions. Android and the Play Store will compare the version code of a new APK to the version code of an installed application to determine if the new APK is indeed an update. The typical approach is to start the version code at `1` and increment it with each production release of your application, though you can choose another convention if you wish. During development, you can leave these alone, but when you move to production, these attributes will matter greatly.

Other Gradle Items of Note

You will always have at least two statements directly in the `android` closure: `compileSdkVersion` and `buildToolsVersion`.

```
android {
    compileSdkVersion 19
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    buildToolsVersion "21.1.2"
}
```

`compileSdkVersion` specifies the API level to be compiled against, usually as a simple API level integer (e.g., 19). An Eclipse project would pull this out of the `project.properties` file in the root of the project directory.

`buildToolsVersion` indicates the version of the Android SDK build tools that you wish to use with this project. While downloading the `android` plugin from Maven Central gives us parts of what is needed, it is not complete. The rest comes from the "Android SDK Build-tools" entry in the SDK Manager:

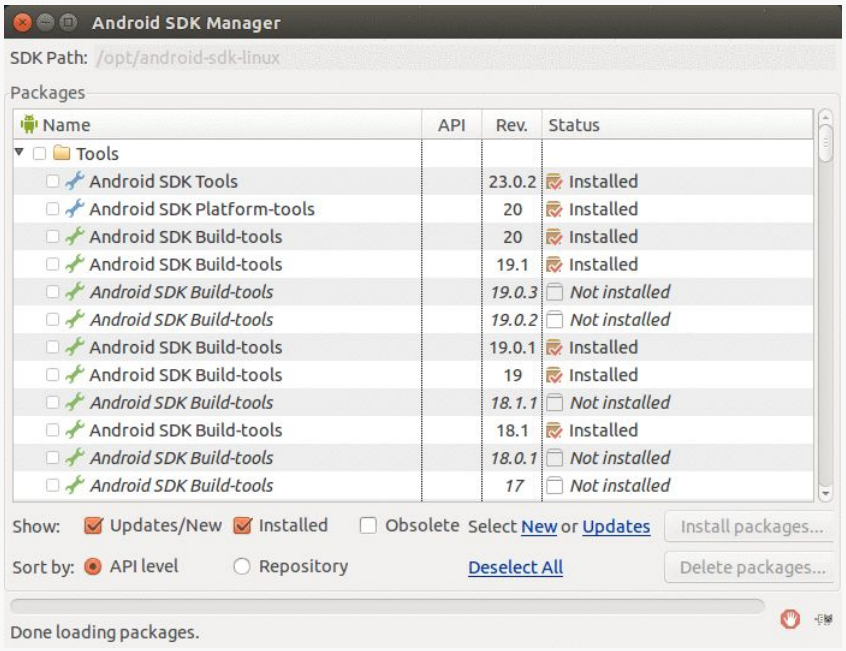


Figure 65: SDK Manager, Showing "Android SDK Build-tools"

Note that the SDK Manager will allow you to download the latest version of the build tools used by Gradle (appearing as 20 in the above screenshot) plus prior versions (e.g., 18.0.1 and 17 in the above screenshot). This corresponds with the `buildToolsVersion` in your `build.gradle` file.

So, your `android` closure could look like:

```
android {
    compileSdkVersion 19
    buildToolsVersion "20.0.0"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    defaultConfig {
        applicationId "com.commonsware.empublite"
        versionCode 1
        versionName "1.0"
        minSdkVersion 15
        targetSdkVersion 18
    }
}
```

Eclipse does not really have the notion of a configurable build tools version, so there is no analogue for `buildToolsVersion` in an Eclipse project.

The Rest of the Manifest

Not everything in the manifest can be overridden in the Gradle build files. Here are a few key items that will always appear in the manifest, regardless of whether this project is to be built by Android Studio, Eclipse, or other means.

An Application For Your Application

In your initial project's manifest, the primary child of the `<manifest>` element is an `<application>` element.

By default, when you create a new Android project, you get a single `<activity>` element inside the `<application>` element:

```
<?xml version="1.0"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.skeleton"
  android:versionCode="1"
  android:versionName="1.0">

  <application>
    <activity
      android:name="Now"
      android:label="Now">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This element supplies `android:name` for the class implementing the activity, `android:label` for the display name of the activity, and (sometimes) an `<intent-filter>` child element describing under what conditions this activity will be displayed. The stock `<activity>` element sets up your activity to appear in the launcher, so users can choose to run it. As we'll see later in this book, you can have several activities in one project, if you so choose.

The `android:name` attribute, in this case, has a bare Java class name (`Now`). Sometimes, you will see `android:name` with a fully-qualified class name (e.g., `com.commonsware.android.skeleton.Now`). Sometimes, you will see a Java class name with a single dot as a prefix (e.g., `.Now`). Both `Now` and `.Now` refer to a Java class that will be in your project's package — the one you declared in the `package` attribute of the `<manifest>` element.

Supporting Multiple Screens

Android devices come with a wide range of screen sizes, from 2.8" tiny smartphones to 46" TVs. Android divides these into four buckets, based on physical size and the distance at which they are usually viewed:

1. Small (under 3")
2. Normal (3" to around 4.5")
3. Large (4.5" to around 10")
4. Extra-large (over 10")

By default, your application will support small and normal screens. It also will support large and extra-large screens via some automated conversion code built into Android.

To truly support all the screen sizes you want, you should consider adding a `<supports-screens>` element to your manifest. This enumerates the screen sizes you have explicit support for. For example, if you are providing custom UI support for large or extra-large screens, you will want to have the `<supports-screens>` element. So, while the starting manifest file works, handling multiple screen sizes is something you will want to think about.

You wind up with an element akin to:

```
<supports-screens
  android:largeScreens="true"
  android:normalScreens="true"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:smallScreens="false"
  android:xlargeScreens="true" />
```

Much more information about providing solid support for all screen sizes, including samples of the `<supports-screens>` element, will be found later in this book as we cover large-screen strategies.

Other Stuff

As we proceed through the book, you will find other elements being added to the manifest, such as:

- `<uses-permission>`, to tell the user that you need permission to use certain device capabilities, such as accessing the Internet
- `<uses-feature>`, to tell Android that you need the device to have certain features (e.g., a camera), and therefore your app should not be installed on devices lacking such features
- `<meta-data>`, for bits of information needed by particular extensions to Android, such as the Google Play Services library.

These and other elements will be introduced elsewhere in the book.

Learning More About Gradle

This book will go into more about Gradle, both in the core chapters and in the trails. But, the focus will be on Gradle for Android, and Gradle itself offers a lot more than that. The Gradle Web site hosts documentation, links to Gradle-specific books, and links to other Gradle educational resources.

At present, the Gradle for Android documentation is limited and mostly appears on the Android tools site. Of note is the top-level page about the new build system, and the Gradle plugin user guide, though both may be out of date compared to the actual tools themselves.

Visit the Trails!

There are a few more chapters in this book getting into more details about the use of Gradle and Gradle for Android.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Gradle and Legacy Projects is for developers who are looking to use Gradle with Eclipse-style projects
- Gradle and Tasks explains how we ask Gradle to do things on our behalf ("tasks"), such as compile our APK for us
- Gradle and the New Project Structure gets into why the project structure changed from what we used with Eclipse, and what capabilities we get as a result, including the ability to configure "build types" and "product flavors"
- Gradle and Dependencies covers more about the "artifacts" mentioned earlier in this chapter, as ways we can get packaged libraries automatically added to our projects via just a couple of lines in our `build.gradle` files

There is also the "Advanced Gradle for Android Tips" chapter for other Gradle topics, and the chapter on manifest merging in Gradle.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #3 - Changing Our Manifest (and Gradle File)

As we build `EmPubLite`, we will need to make a number of changes to our project's manifest. In this tutorial, we will take care of a couple of these changes, to show you how to manipulate the `AndroidManifest.xml` file. Future tutorials will make yet more changes.

Android Studio users will also get their first chance to work with the `build.gradle` file.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Some Notes About Relative Paths

In these tutorials, you will see references to relative paths, like `AndroidManifest.xml`, `res/layout/`, and so on.

Eclipse users should interpret these paths as being relative to the project root directory.

Android Studio users should interpret these paths as being relative to the `app/src/main/` directory within the project, except as otherwise noted. So, for example, Step

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

#1 below will ask you to open `AndroidManifest.xml` — that file can be found in `app/src/main/AndroidManifest.xml` from the project root.

Step #1: Supporting Screens

Our application will restrict its supported screen sizes. Tablets make for ideal ebook readers. Phones can also be used, but the smaller the phone, the more difficult it will be to come up with a UI that will let the user do everything that is needed, yet still have room for more than a sentence or two of the book at a time.

We will get into screen size strategies and their details later in this book. For the moment, though, we will add a `<supports-screens>` element to keep our application off "small" screen devices (under 3" diagonal size).

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Double-click on `AndroidManifest.xml` in your project explorer.

As a child of the root `<manifest>` element, add a `<supports-screens>` element as follows:

```
<supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"
    android:xlargeScreens="true"/>
```

Eclipse

In the Package Explorer view in Eclipse, find the `AndroidManifest.xml` entry and double-click on it.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

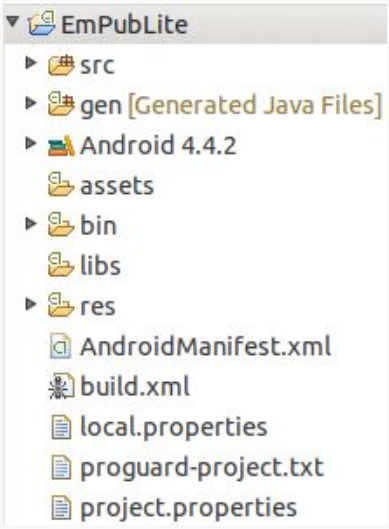


Figure 66: Eclipse Package Explorer, Showing EmPubLite

Double-clicking on the file will bring the file up in Eclipse's default editor for that type of file. In the case of `AndroidManifest.xml`, this will be a structured editor for manifest settings:

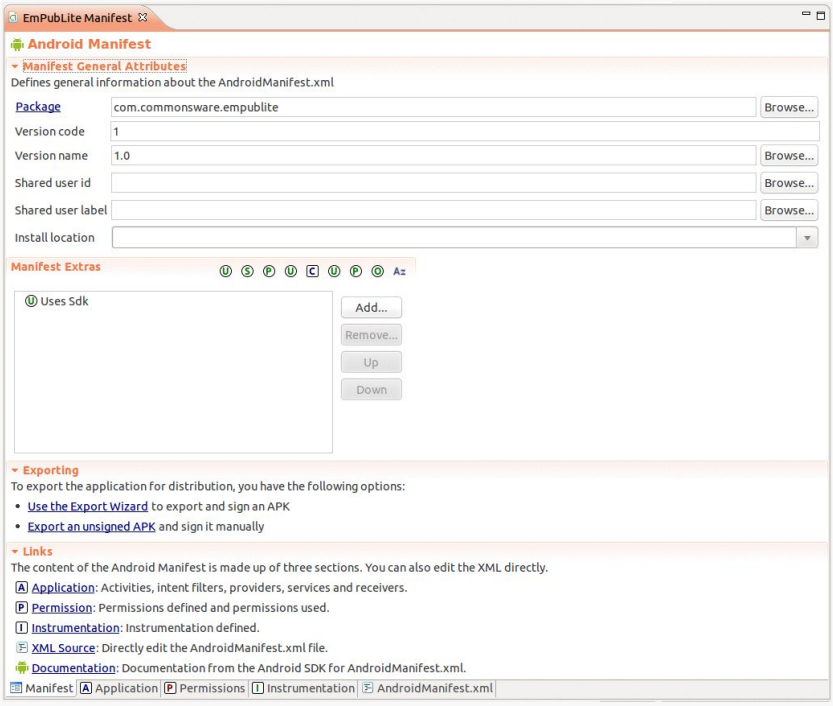


Figure 67: Eclipse Manifest Editor

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You will notice that there is a series of sub-tabs at the bottom of the editor, labeled "Manifest", "Application", "Permissions", and so on. These allow you to adjust different portions of the manifest file. The right-most sub-tab, "AndroidManifest.xml", allows you to edit the raw XML of this file directly, if you so choose. This is a fairly typical pattern with the Eclipse editors: one or more sub-tabs providing a structured way of editing the data, and the right-most sub-tab providing raw access to the underlying XML.

In the "Manifest Extras" area of the "Manifest" sub-tab in our open manifest editor, click the "Add..." button to the right of the extras list, to bring up a dialog of what sort of extras we can add:

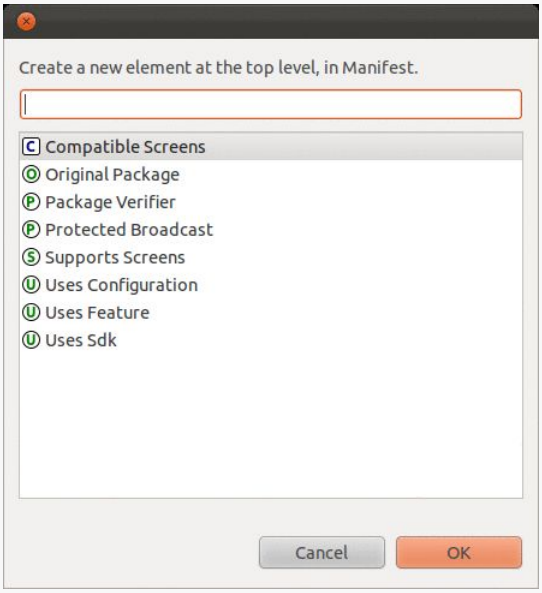


Figure 68: Eclipse Manifest Extras Options

Click on "Supports Screens", then click "OK" to close the dialog and add a "Supports Screens" entry in the "Manifest Extras" list. That entry will be pre-selected by the editor, showing the available configuration options on the right:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

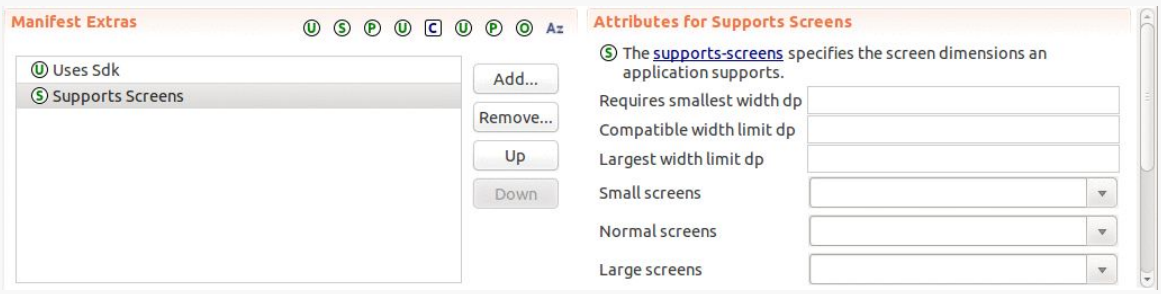


Figure 69: Eclipse Supports Screens Options

Note that the attributes list on the right may have vertical scrollbar, as there are several things we can stipulate on the `<supports-screens>` element, and not all can fit on the editor at once given the editor's design.

Using that scrollbar as needed, toggle the "Small screens" value to `false` and the "Normal screens", "Large screens", and "Xlarge screens" values to `true`:

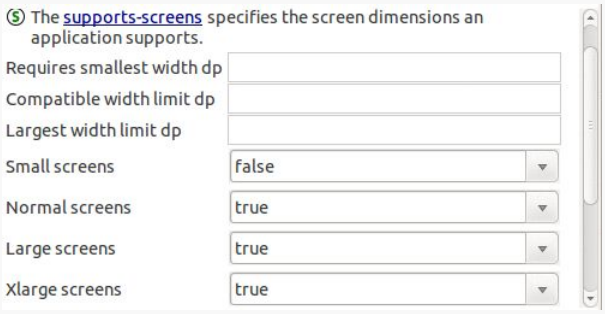


Figure 70: Eclipse Supports Screens Options, Adjusted

Then you can save the file, via the main menu, the Save toolbar icon, or `<Ctrl>-<S>`.

Step #2: Adding our Minimum and Target SDK Versions

We also need to teach our IDE our minimum SDK version (how old a version of Android we will support) and our target SDK version (what version of Android we were thinking of when writing our code).

Eclipse users will accomplish this by adding a `<uses-sdk>` element to the manifest. Android Studio users *could* do the same, but instead we will follow the Android Studio convention and add those values to `build.gradle`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Double-click on `app/build.gradle`, off of the project root, in the Project Explorer.

You should see in there a `defaultConfig` closure that looks like:

```
    defaultConfig {
        applicationId "com.commonsware.empublite"
        versionCode 1
        versionName "1.0"
    }
```

Add the `minSdkVersion 15` and `targetSdkVersion 18` statements to it, so that it looks like:

```
    defaultConfig {
        applicationId "com.commonsware.empublite"
        versionCode 1
        versionName "1.0"
        minSdkVersion 15
        targetSdkVersion 18
    }
```

You may get a yellow banner at the top of the editor, indicating that a "project sync" is requested. If you do, click the "Sync Now" link in that banner to synchronize the `*.iml` files with the changes you made to this `build.gradle` file.

Eclipse

You should already have the `AndroidManifest.xml` file open in Eclipse from Step #1 of this tutorial.

In the "Manifest Extras" area of the "Manifest" sub-tab in our open manifest editor, see if you already have a "Uses Sdk" entry. If yes, click it to fill in the details pane to the right. If you do not have a "Uses Sdk" entry, click the "Add..." button to the right of the extras list, choose "Uses Sdk", then click "OK" to close the dialog and add a "Supports Screens" entry in the "Manifest Extras" list:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

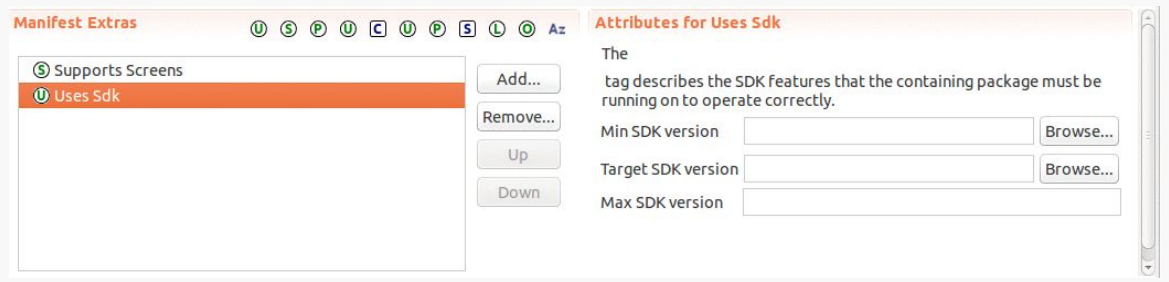


Figure 71: Eclipse Manifest Extras, Showing "Uses Sdk"

In the "Min SDK version" field, fill in `15`, meaning that we will support Android 4.0.3 and higher.

In the "Target SDK version" field, fill in `18`.

Leave the "Max SDK version" field blank, as that is unused in Android and most marketplaces.

Then you can save the file, via the main menu, the Save toolbar icon, or `<Ctrl>-<S>`.

In Our Next Episode…

… we will make some changes to the resources of our tutorial project

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Some Words About Resources

It is quite likely that by this point in time, you are "chomping at the bit" to get into actually writing some code. This is understandable. That being said, before we dive into the Java source code for our stub project, we really should chat briefly about resources.

Resources are static bits of information held outside the Java source code. Resources are stored as files under the `res/` directory in your Android project layout (whether that is in the project root for Eclipse or in the `main/` sourceset for Android Studio). Here is where you will find all your icons and other images, your externalized strings for internationalization, and more.

These are not only separate from the Java source code because they are different in format. They are separate because you can have *multiple* definitions of a resource, to use in different circumstances. For example, with internationalization, you will have strings for different languages. Your Java code will be able to remain largely oblivious to this, as Android will choose the right resource to use, from all candidates, in a given circumstance (e.g., choose the Spanish string if the device's locale is set to Spanish).

We will cover all the details of these resource sets later in the book. Right now, we need to discuss the resources in use by our stub project, plus one more.

This chapter will refer to the `res/` directory. Android Studio users will find that in the `app/src/main/` directory of their project, while Eclipse users will find it in the project root directory.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

String Theory

Keeping your labels and other bits of text outside the main source code of your application is generally considered to be a very good idea. In particular, it helps with internationalization (I18N) and localization (L10N). Even if you are not going to translate your strings to other languages, it is easier to make corrections if all the strings are in one spot instead of scattered throughout your source code.

Plain Strings

Generally speaking, all you need to do is have an XML file in the `res/values` directory (typically named `res/values/strings.xml`), with a `resources` root element, and one child `string` element for each string you wish to encode as a resource. The `string` element takes a `name` attribute, which is the unique name for this string, and a single text element containing the text of the string:

```
<resources>
  <string name="quick">The quick brown fox...</string>
  <string name="laughs">He who laughs last...</string>
</resources>
```

One tricky part is if the string value contains a quote or an apostrophe. In those cases, you will want to escape those values, by preceding them with a backslash (e.g., `These are the times that try men\'s souls`). Or, if it is just an apostrophe, you could enclose the value in quotes (e.g., `"These are the times that try men's souls."`).

For example, a project's `strings.xml` file could look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

  <string name="app_name">EmPubLite</string>
  <string name="hello_world">Hello world!</string>

</resources>
```

We can reference these string resources from various locations, in our Java source code and elsewhere. For example, the `app_name` string resource often is used in the `AndroidManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.empublite"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"
    android:xlargeScreens="true"/>

  <uses-sdk
    android:minSdkVersion="15"
    android:targetSdkVersion="18"/>

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name="EmPubLiteActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Here, the `android:label` attribute of the `<application>` element refers to the `app_name` string resource. This will appear in a few places in the application, notably in the list of installed applications in Settings. So, if you wish to change how your application's name appears in these places, simply adjust the `app_name` string resource to suit.

The syntax `@string/app_name` tells Android "find the string resource named `app_name`". This causes Android to scan the appropriate `strings.xml` file (or any other file containing string resources in your `res/values/` directory) to try to find `app_name`.

Styled Text

Many things in Android can display rich text, where the text has been formatted using some lightweight HTML markup: `<b>`, `<i>`, and `<u>`. Your string resources support this, simply by using the HTML tags as you would in a Web page:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<resources>
  <string name="b">This has <b>bold</b> in it.</string>
  <string name="i">Whereas this has <i>italics</i>!</string>
</resources>
```

CDATA. CDATA Run. Run, DATA, Run.

Since a strings resource XML file is an XML file, if your message contains <, >, or & characters (other than the formatting tags listed above), you will need to use a `CDATA` section:

```
<string name="report_body">
<![CDATA[
<html>
<body>
<h1>TPS Report for: {{reportDate}}</h1>
<p>Here are the contents of the TPS report:</p>
<p>{{message}}</p>
<p>If you have any questions regarding this report, please
do <b>not</b> ask Mark Murphy.</p>
</body>
</html>
]]>
  </string>
```

The Directory Name

Our string resources in our stub project are in the `res/values/strings.xml` file. Since this directory name (`values`) has no suffixes, the string resources in that directory will be valid for any sort of situation, including any locale for the device. We will need additional directories, with distinct `strings.xml` files, to support other languages. We will cover how to do that later in this book.

Editing String Resources

If you double-click on a string resource file, like `res/values/strings.xml`, in Android Studio, you are presented the XML and edit it that way.

When you double-click on a string resource file in Eclipse, you will be greeted with a list of all the string resources that have been defined:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

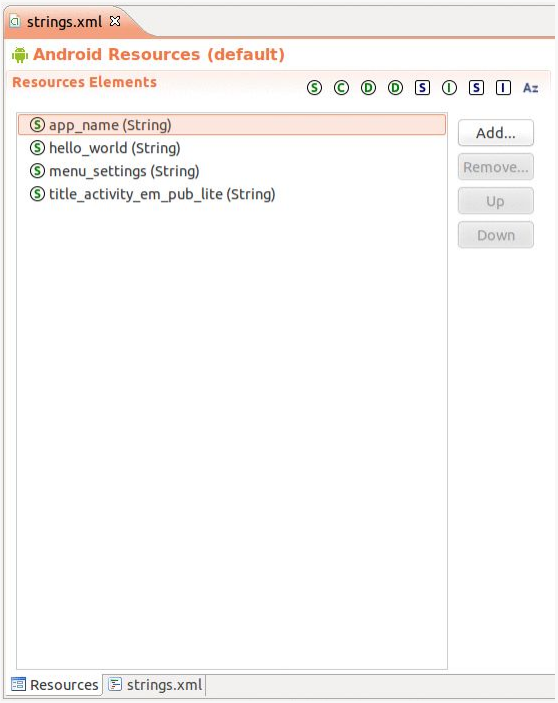


Figure 72: Eclipse, Showing String Resources

Clicking on a resource allows you to edit its name and value:

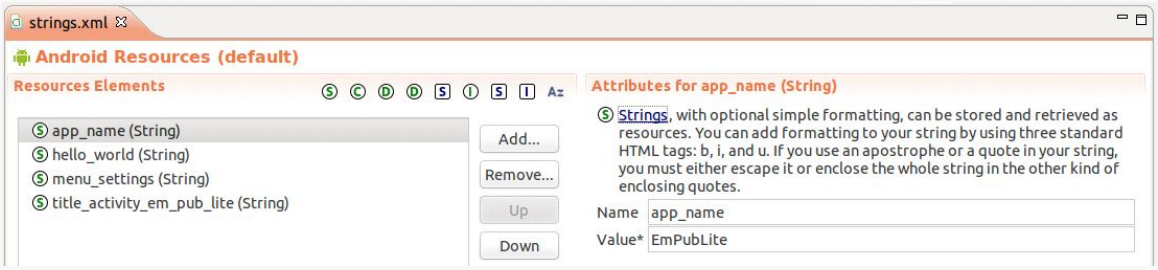


Figure 73: Eclipse, Editing Existing String Resources

Clicking the "Add..." button to the right of the list of strings brings up a dialog where you can add another resource to this file, typically a string:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

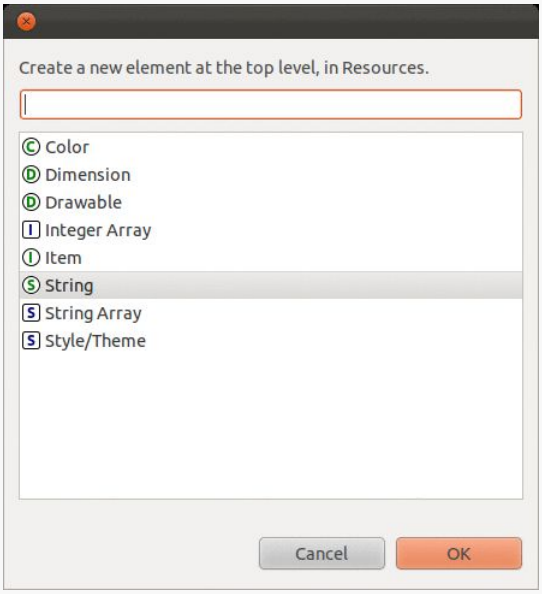


Figure 74: Eclipse, Add String Resource Dialog

Choosing "String" in that dialog and clicking OK will add another (empty) string resource to the list, where you can fill in the name and value.

You can always click on the `strings.xml` sub-tab to bring up an XML editor on the string resources if you prefer.

Got the Picture?

Android supports images in the PNG, JPEG, and GIF formats. GIF is officially discouraged, however; PNG is the overall preferred format. Android also supports some proprietary XML-based image formats, though we will not discuss those at length until later in the book.

The default directory for these so-called drawable resources is `res/drawable/`. Any images found in there can be referenced from Java code or from other places (such as the manifest), regardless of device characteristics.

However, your stub project does not have a `res/drawable/` directory.

Instead, it has directories like `res/drawable-mdpi/` and `res/drawable-hdpi/`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

These refer to distinct resource sets. The suffixes (e.g., `-mdpi`, `-hdpi`) are filters, indicating under what circumstances the images stored in those directories should be used. Specifically, `-ldpi` indicates images that should be used on devices with low-density screens (around 120 dots-per-inch, or "dpi"). The `-mdpi` suffix indicates resources for medium-density screens (around 160dpi), `-hdpi` indicates resources for high-density screens (around 240dpi). `-xhdpi` indicates resources for extra-high-density screens (around 320dpi), `--xxhdpi` indicates extra-extra-high-density screens (around 480dpi), `-xxxhdpi` indicates extra-extra-extra-high-density screens (around 640dpi), and so on.

Inside each of those directories, you will see an `ic_launcher.png` file (along with perhaps other icons). This is the stock icon that will be used for your application in the home screen launcher. Each of the images is of the same icon, but the higher-density icons have more pixels. The objective is for the image to be roughly the same physical size on every device, using higher densities to have more detailed images.

For example, our `EmPubLite` tutorial project has directories like `res/drawable-hdpi/`, `res/drawable-xhdpi/`, and `res/drawable-mdpi/`. Each could contain a stock launcher icon (`ic_launcher.png`) tailored for that density.

Our `AndroidManifest.xml` file then references our `ic_launcher` icon in the `<application>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.empublite"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"
    android:xlargeScreens="true"/>

  <uses-sdk
    android:minSdkVersion="15"
    android:targetSdkVersion="18"/>

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name="EmPubLiteActivity"
      android:label="@string/app_name">
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Note that the manifest simply refers to `@drawable/ic_launcher`, telling Android to find a drawable resource named `ic_launcher`. The resource reference does not indicate the file type of the resource — there is no `.png` in the resource identifier. This means you cannot have `ic_launcher.png` and `ic_launcher.jpg` in the same project, as they would both be identified by the same identifier. You will need to keep the "base name" (filename sans extension) distinct for all of your images.

Also, the `@drawable/ic_launcher` reference does not mention what screen density to use. That is because *Android* will choose the right screen density to use, based upon the device that is running your app. You do not have to worry about it explicitly, beyond having multiple copies of your icon.

If Android detects that the device has a screen density for which you lack an icon, Android will take the next-closest one and scale it.

Getting Android Drawables

You may be a graphic designer. Or, you may know a graphic designer. In those cases, you can create your own icons, ideally following Google's design guidelines for iconography.

If you are not a graphic designer and do not have ready access to one, you will need to come up with your drawable resources by other means. There are plenty of icon libraries available from third parties, but the following sections outline some of Google's solutions for putting icons in your app.

Android Icon Set Wizard

Both Android Studio and Eclipse offer an icon set wizard. This wizard is designed to take a starter image and give you icons, in a variety of densities, that use that image for a particular image role, such as your home screen launcher icon (the `ic_launcher.png` file we saw earlier in this chapter).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We will explore how to use this wizard, for both Android Studio and Eclipse, in the next tutorial.

Android Asset Studio

The same basic functionality found in the icon set wizard is available outside any IDE (but inside a Chrome browser) in the form of the Android Asset Studio. As with the icon set wizard, you can choose a type of icon (e.g., launcher icons):

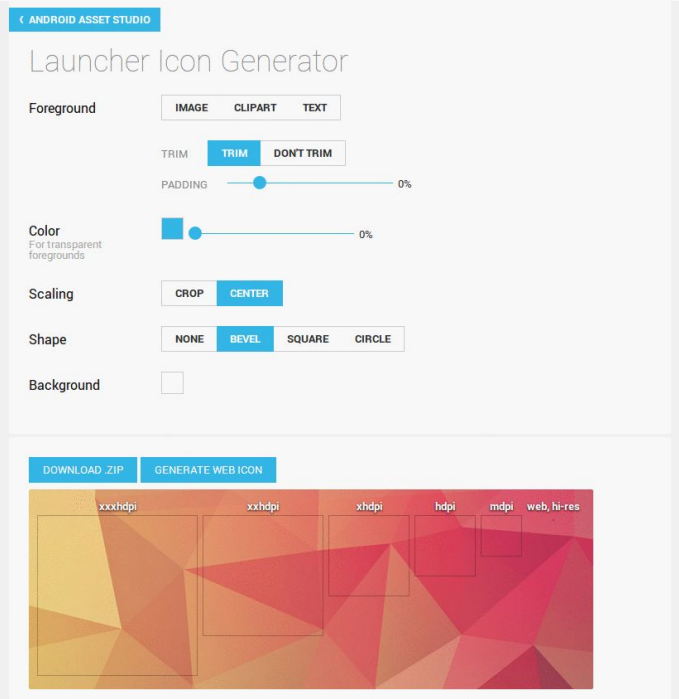


Figure 75: Android Asset Studio, Launcher Icon Page

Then you can specify the source of the base image (uploaded file, canned clipart, or free-form text) and other configuration data. The resulting images, in various densities, can be downloaded at the bottom of the page:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

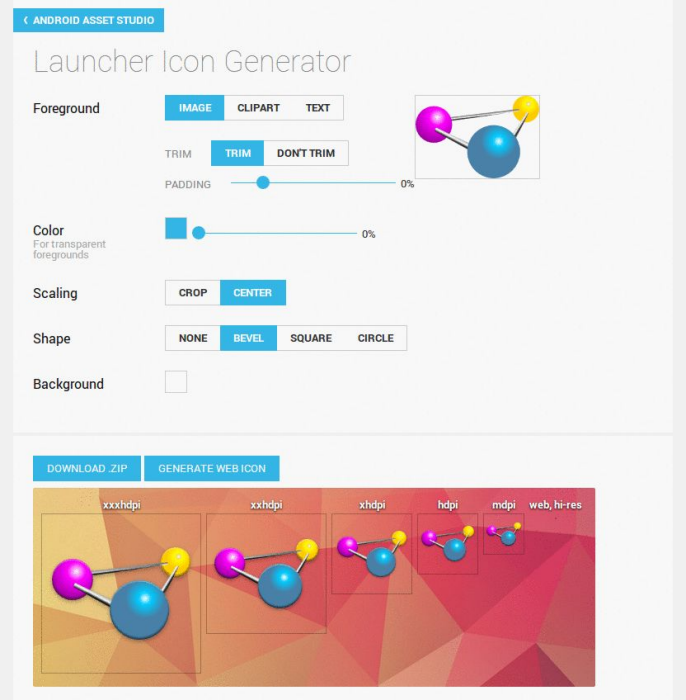


Figure 76: Android Asset Studio, Launcher Icon Page, with Icons

The Action Bar Icon Pack

Much of the clipart used by the icon set wizard and Android Asset Studio comes from Google's Action Bar Icon Pack, which is a set of icons, at various densities, designed to be used as icons for toolbar-style buttons in the action bar. We will talk more about what the action bar is and how you use it later in the book, and in a subsequent tutorial we will download and use icons from the Action Bar Icon Pack.

Editing Existing Drawable Resources

Neither Android Studio nor Eclipse ships with any sort of image editor that you could use for PNG and JPEG files. Hence, you will find yourself editing these images using other tools outside of your IDE.

Dimensions

Dimensions are used in several places in Android to describe distances, such as a widget's size. There are several different units of measurement available to you:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. `px` means hardware pixels, whose size will vary by device, since not all devices have the same screen density
2. `in` and `mm` for inches and millimeters, respectively, based on the actual size of the screen
3. `pt` for points, which in publishing terms is 1/72nd of an inch (again, based on the actual physical size of the screen)
4. `dip` (or `dp`) for density-independent pixels — one `dip` equals one hardware pixel for a ~160dpi resolution screen, but one `dip` equals two hardware pixels on a ~320dpi screen
5. `sp` for scaled pixels, where one `sp` equals one `dip` for normal font scale levels, increasing and decreasing as needed based upon the user's chosen font scale level in Settings

Dimension resources, by default, are held in a `dimens.xml` file in the `res/values/` directory that also holds your strings.

To encode a dimension as a resource, add a `dimen` element to `dimens.xml`, with a `name` attribute for your unique name for this resource, and a single child text element representing the value:

```
<resources>
  <dimen name="thin">10dip</dimen>
  <dimen name="fat">1in</dimen>
</resources>
```

In a layout, you can reference dimensions as `@dimen/...`, where the ellipsis is a placeholder for your unique name for the resource (e.g., `thin` and `fat` from the sample above). In Java, you reference dimension resources by the unique name prefixed with `R.dimen.` (e.g., `Resources.getDimension(R.dimen.thin)`).

While our stub project does not use dimension resources, we will be seeing them soon enough.

Editing Dimension Resources

As with most types of XML resources, Android Studio just has you edit the XML directly, when you double-click on the resource in the project explorer.

In Eclipse, much like editing string resources, when you double-click on a dimension resource file (e.g., `res/values/dimens.xml`), you will be presented with a list of existing dimensions. Clicking on one will let you change its definition:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

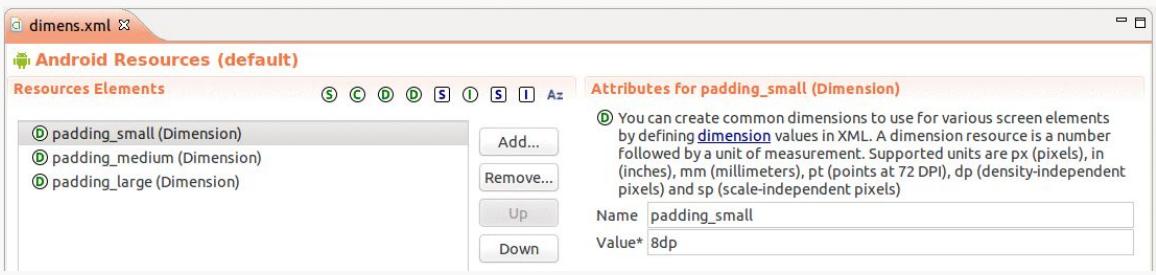


Figure 77: Eclipse, Editing Existing Dimension Resources

Clicking the "Add..." button to the right of the list of dimensions brings up a dialog where you can add another resource to this file, typically a dimension. Choosing "Dimension" and clicking "OK" will add an empty dimension resource to the file, for which you can supply the name and value.

And, as always, you can click on a sub-tab with the name of your file (e.g., `dimens.xml`) to bring up an XML editor on that resource's content:

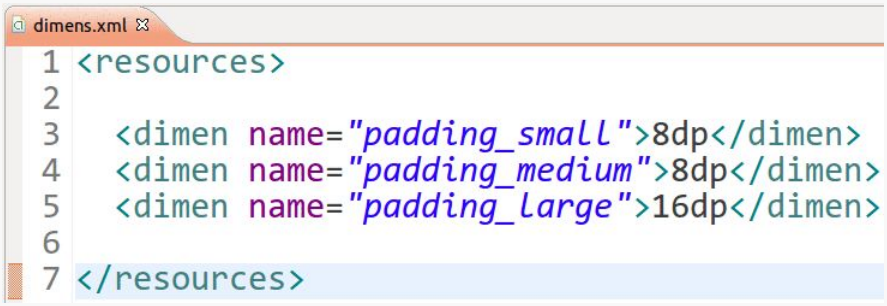


Figure 78: Eclipse, Dimension Resources in XML Editor

The Resource That Shall Not Be Named… Yet

Your stub project also has a `res/layout/` directory, in addition to the ones described above. That is for UI layouts, describing what your user interface should look like. We will get into the details of that type of resource as we start examining our user interfaces in an upcoming chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #4 - Adjusting Our Resources

Our `EmPubLite` project has some initial resources. However, the defaults are not what we want for the long term. So, in addition to adding new resources in future tutorials, we will fix the ones we already have in this tutorial.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Changing the Name

Our application shows up everywhere as "EmPubLite":

- In the title bar of our activity
- As the caption under our icon in the home screen launcher
- In the Application list in the Settings app
- And so on

We should change that to be "EmPub Lite", adding a space for easier reading, and to illustrate that this is a "lite" version of the full `EmPub` application.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio

Double-click on the `res/values/strings.xml` file in your project explorer.

In the XML editor for the string resources, you will find an element that looks like:

```
<string name="app_name">EmPubLite</string>
```

Change the text node in this element to `EmPub Lite`. Then save your changes, giving you something like:

```
<resources>

  <string name="app_name">EmPub Lite</string>
  <string name="hello_world">Hello world!</string>

</resources>
```

Eclipse

In the Package Explorer, open up the `res/values/` folder — you should see a `strings.xml` file in there:

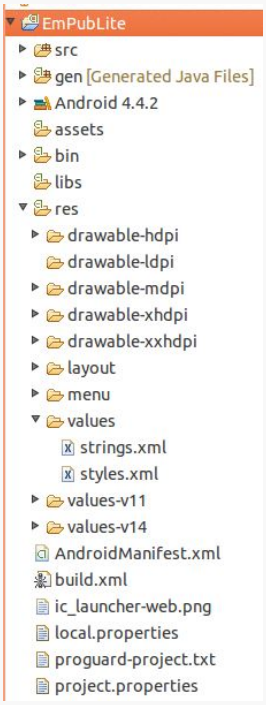


Figure 79: Eclipse Package Explorer, Showing EmPubLite

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Double-click on `strings.xml` to open it in the string resources editor:

Figure 80: Eclipse String Resources Editor

This shows a list of the defined string resources (denoted by the green S in the circle) in this file.

Click the `app_name` resource, to bring up its details on the right:

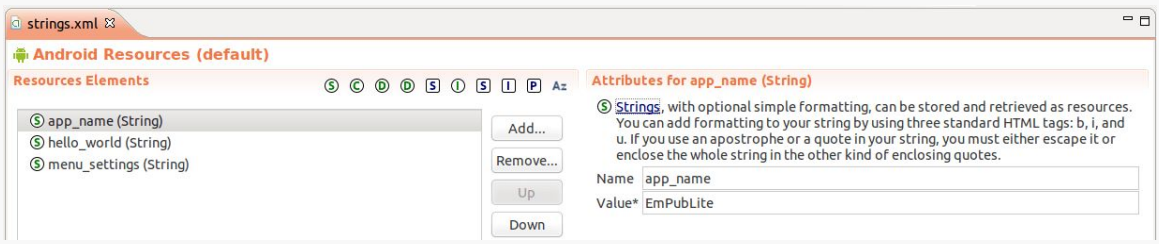


Figure 81: Eclipse String Resources Editor with Details

The `app_name` name for the resource is fine, as that is how this string is referenced from the manifest. Change the value to be "EmPub Lite" (adding the space).

Step #2: Changing the Icon

The build tools provide us with a stock icon to use for the launcher — the actual image used varies by Android tools release. However, we can change it to something else. For example, we could use the icon portion of the CommonsWare logo:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

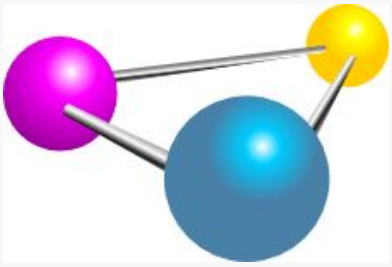


Figure 82: CommonsWare

First, download the original image and save it somewhere on your development machine.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Click on the `app/` entry in the project explorer, then choose File > New... from the main menu. This should pop up a small window for you to choose something new to create:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

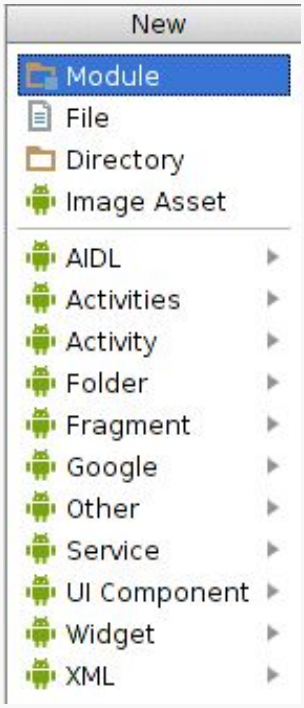


Figure 83: Android Studio New Item Dialog

In there, choose "Image Asset". This will bring up an "Asset Studio" wizard for creating images:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

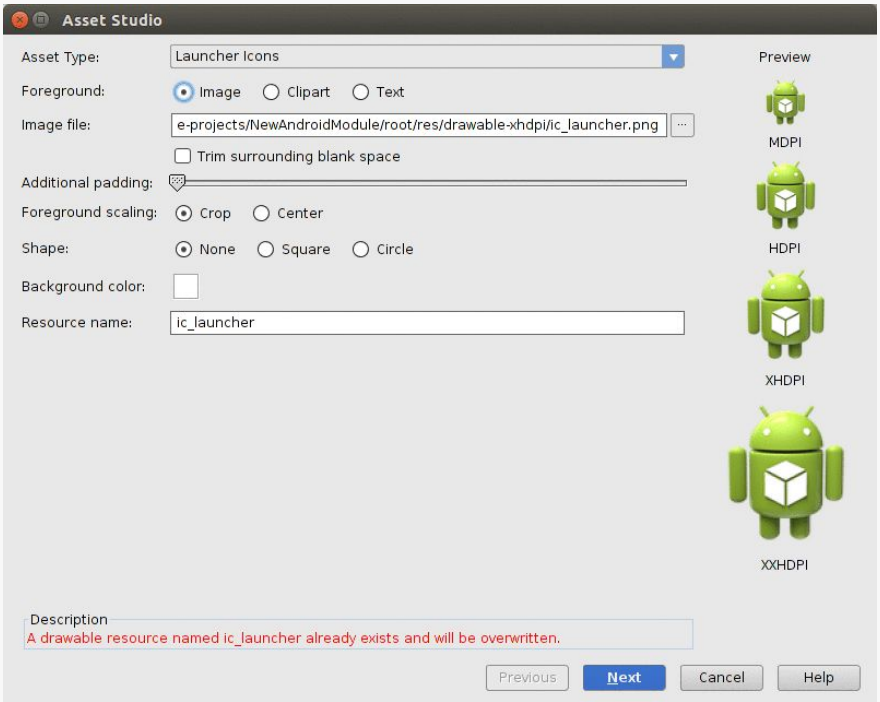


Figure 84: Android Studio "Asset Studio" Wizard, First Page

Ensure that the "Asset Type" drop-down is set to "Launcher Icons", and the "Foreground" option is set to "Image". Then, use the "..." button next to the "Image file" field to choose the image that you downloaded above.

Then, set the "Foreground scaling" to "Center", leaving you with a wizard page that should resemble:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

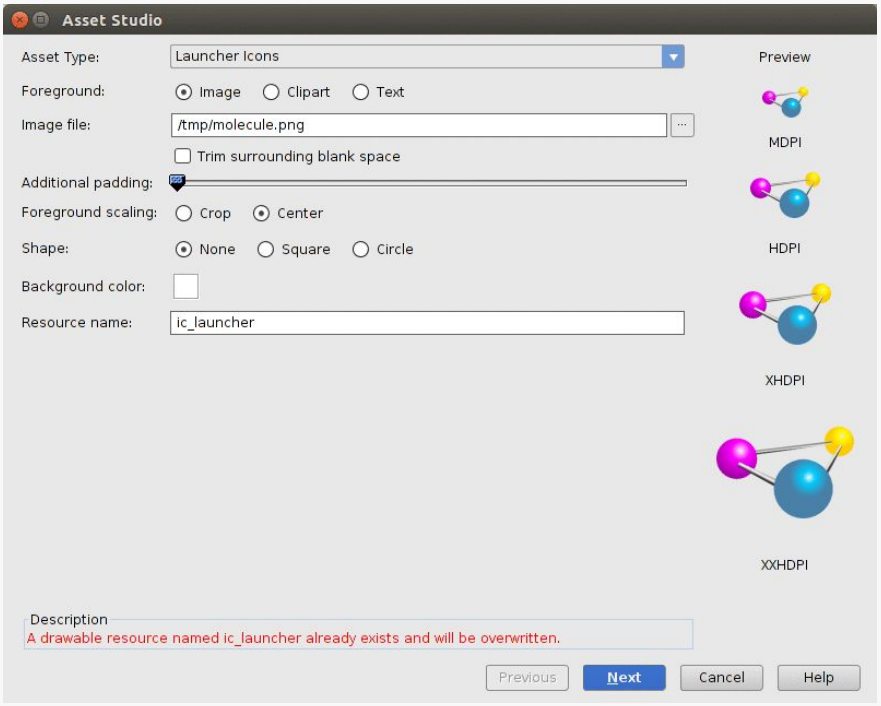


Figure 85: Android Studio "Asset Studio" Wizard, First Page, Filled In

Click "Next" to bring up the second page of the wizard, which is there to confirm where you want the images written and what images will be created:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

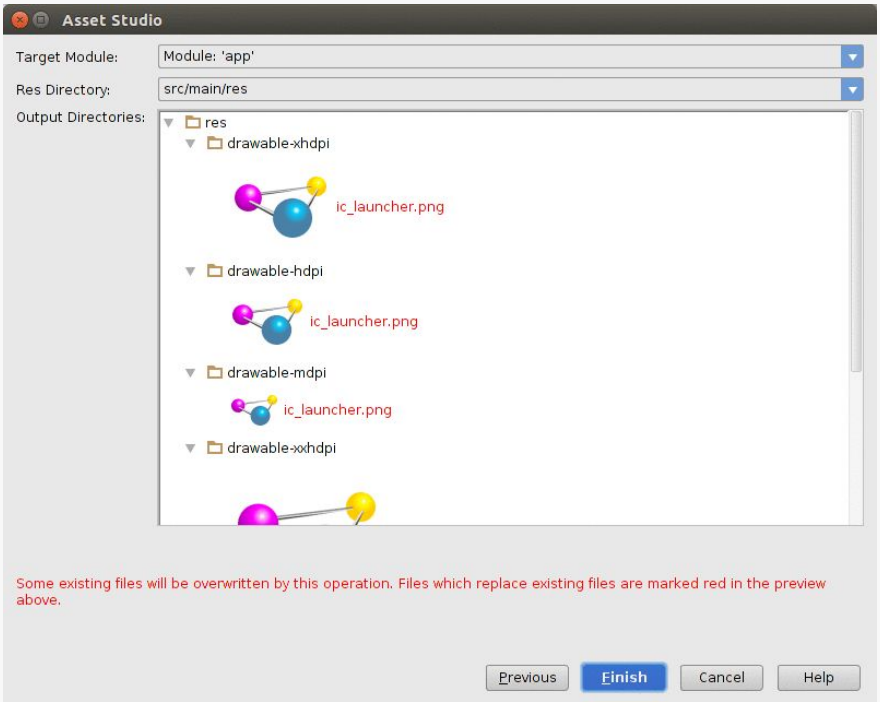


Figure 86: Android Studio "Asset Studio" Wizard, Second Page

Click "Finish", and your original icons will be replaced with ones based on your chosen image.

Eclipse

From the Eclipse main menu, choose File > New > Other > Android. In the resulting dialog, choose "Android Icon Set" and press Next.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

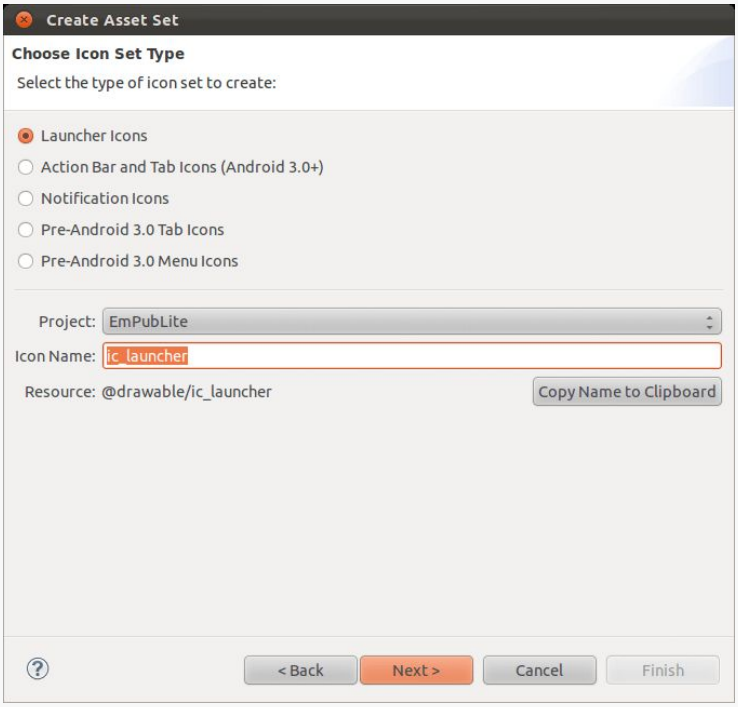


Figure 87: Eclipse Icon Set Wizard, First Page

The defaults on the first page of the icon set wizard are to create launcher icons, with a file base name of `ic_launcher`, to be added to the `EmPubLite` project. If the values that you see in the wizard do not match that, adjust the wizard, then press Next.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

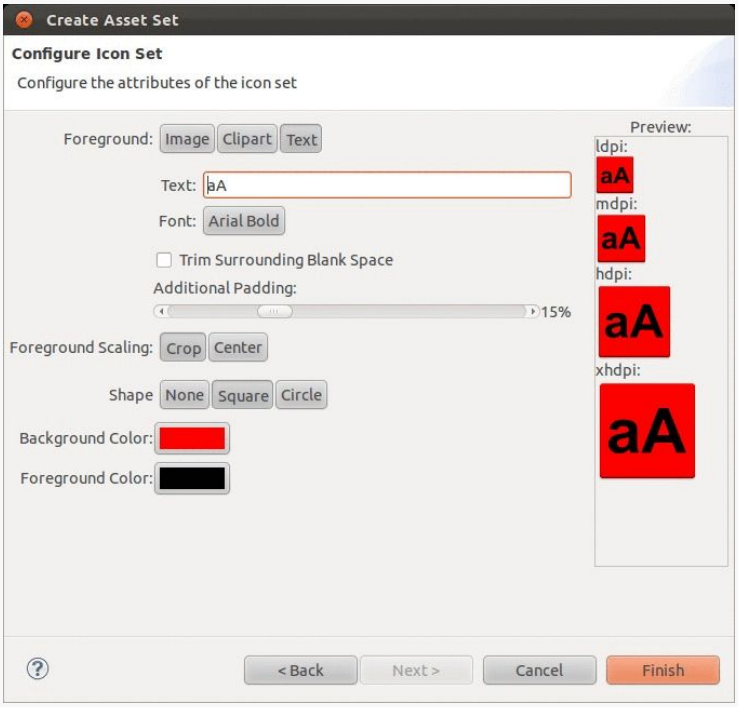


Figure 88: Eclipse Icon Set Wizard, Second Page

In the second page of the icon set wizard, click the "Image" button in the "Foreground" row. This will change the wizard slightly, giving you a space to supply the path to some image:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

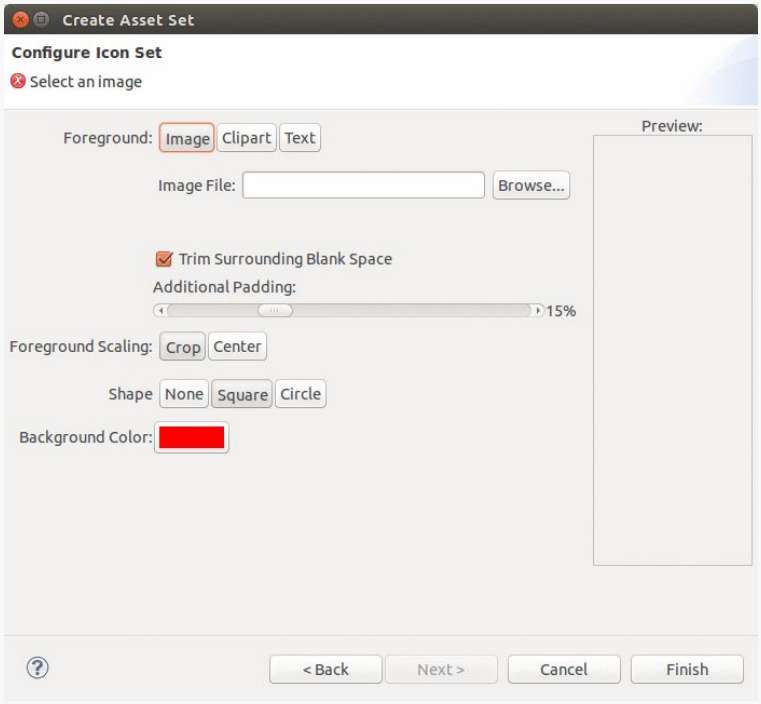


Figure 89: Eclipse Icon Set Wizard, Second Page, Image Mode

Click the "Browse..." button and open the `molecule.png` file you downloaded above. That will display the results in the wizard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

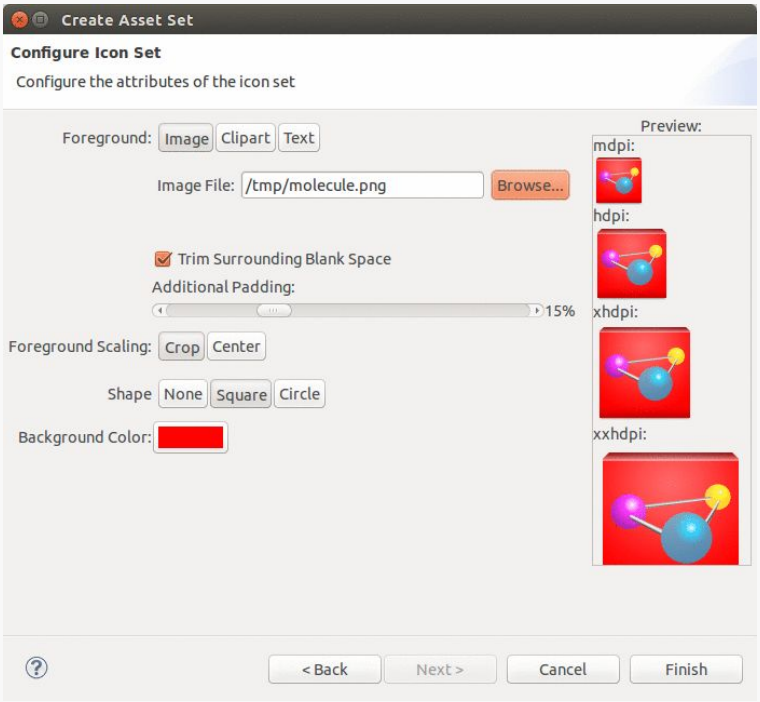


Figure 90: Eclipse Icon Set Wizard, Second Page, Image Mode, Showing Molecule

Click the "None" button in the "Shape" row, to remove the square background. Then, click Finish. You will be prompted for whether you want to overwrite the existing images — click "Yes to All".

You may wind up with a bunch of error markers on your project for all of the new images in the Package Explorer. If this occurs, choose Project > Clean from the Eclipse main menu, ensure that EmPubLite is checked in the project list, and choose OK. This should get rid of those error markers.

Step #3: Running the Result

If you run the resulting app, you will see that it shows up with the new name and icon, such as in the launcher:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

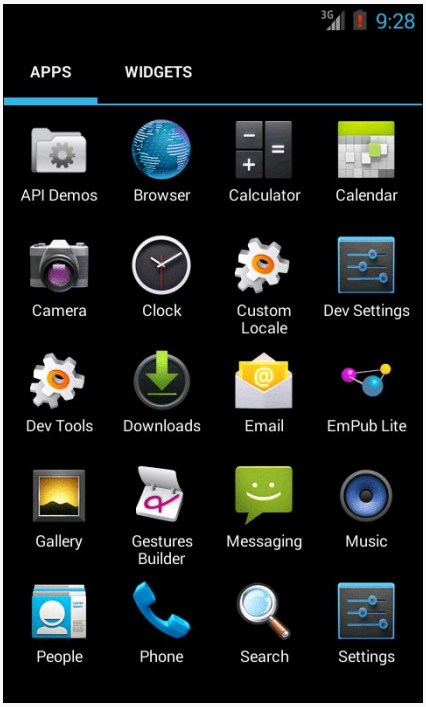


Figure 91: EmPubLite with New Icons

However, Eclipse users may encounter some problems in running the result. When you wish to run an Android project from Eclipse, you must pay close attention to what part of the Eclipse UI has the focus. The focus *cannot* be on an editor for a resource. So, for example, had you gone back to the string resource editor, done some changes there, then attempted to run the project, nothing would have happened.

Instead, the focus has to be pretty much anywhere else for the Run option in the toolbar to work:

- On the manifest
- On some Java code
- On the Package Explorer

This is a bug, one that will hopefully get fixed someday. However, since that preceding sentence has been in this book for a couple of years, it is unlikely that the bug will be addressed any time soon.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In Our Next Episode…

… we will add a progress indicator to the UI of our tutorial project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Theory of Widgets

There is a decent chance that you have already done work with widget-based UI frameworks. In that case, much of this chapter will be review, though checking out the section on the absolute positioning anti-pattern should certainly be worthwhile.

There is a chance, though, that your UI background has come from places where you have not been using a traditional widget framework, where either you have been doing all of the drawing yourself (e.g., game frameworks) or where the UI is defined more in the form of a document (e.g., classic Web development). This chapter is aimed at you, to give you some idea of what we are talking about when discussing the notion of widgets and containers.

What Are Widgets?

Wikipedia has a nice definition of a widget:

> In computer programming, a widget (or control) is an element of a graphical user interface (GUI) that displays an information arrangement changeable by the user, such as a window or a text box. The defining characteristic of a widget is to provide a single interaction point for the direct manipulation of a given kind of data. In other words, widgets are basic visual building blocks which, combined in an application, hold all the data processed by the application and the available interactions on this data.

(quote from the 7 March 2014 version of the page)

Take, for example, this Android screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

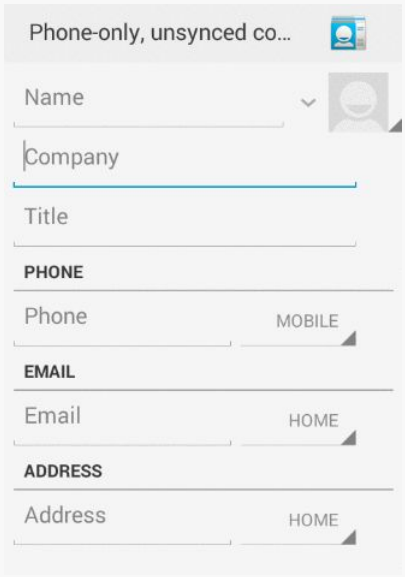


Figure 92: A Sample Android Screen

Here, we see:

- some text, like “Phone-only, unsynced co...” and “PHONE”
- an icon of a contact “Rolodex” card
- some data entry fields with hints like “Name” and “Company”
- some “spinner” drop-down lists (the items with the arrowheads pointing southeast)
- some gray divider lines

Everything listed above is a widget. The user interface for most Android screens (“activities”) is made up of one or more widgets.

This does not mean that you cannot do your own drawing. In fact, all the existing widgets are implemented via low-level drawing routines, which you can use for everything from your own custom widgets to games.

This also does not mean that you cannot use Web technologies. In fact, we will see later in this book a widget designed to allow you to embed Web content into an Android activity.

However, for most non-game applications, your Android user interface will be made up of several widgets.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Size, Margins, and Padding

Widgets have some sort of size, since a zero-pixel-high, zero-pixel-wide widget is not especially user-friendly. Sometimes, that size will be dictated by what is inside the widget itself, such as a label (`TextView`) having a size dictated by the text in the label. Sometimes, that size will be dictated by the size of whatever holds the widget (a "container", described in the next section), where the widget wants to take up all remaining width and/or height. Sometimes, that size will be a specific set of dimensions.

Widgets can have margins. As with CSS, margins provide separation between a widget and anything adjacent to it (e.g., other widgets, edges of the screen). Margins are really designed to help prevent widgets from running right up next to each other, so they are visually distinct. Some developers, however, try to use margins as a way to hack "absolute positioning" into Android, which is an anti-pattern that we will examine later in this chapter.

Widgets can have padding. As with CSS, padding provides separation between the contents of a widget and the widget's edges. This is mostly used with widgets that have some sort of background, like a button, so that the contents of the widget (e.g., button caption) does not run right into the edges of the button, once again for visual distinction.

What Are Containers?

Containers are ways of organizing multiple widgets into some sort of structure. Widgets do not naturally line themselves up in some specific pattern — we have to define that pattern ourselves.

In most GUI toolkits, a container is deemed to have a set of children. Those children are widgets, or sometimes other containers. Each container has its basic rule for how it lays out its children on the screen, possibly customized by requests from the children themselves.

Common container patterns include:

- put all children in a row, one after the next
- put all children in a column, one below the next
- arrange the children into a table or grid with some number of rows and columns

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- anchor the children to the sides of the container, according to requests made by those children
- anchor the children to other children in the container, according to requests made by those children
- stack all children, one on top of the next
- and so on

In the sample activity above, the dominant pattern is a column, with things laid out from top to bottom. Some of those things are rows, with contents laid out left to right. However, as it turns out, the area with most of those widgets is scrollable.

Android supplies a handful of containers, designed to handle most common scenarios, including everything in the list above. You are also welcome to create your own custom containers, to implement business rules that are not directly supported by the existing containers.

Note that containers also have size, padding, and margins, just as widgets do.

The Absolute Positioning Anti-Pattern

You might wonder why all of these containers and such are necessary. After all, can't you just say that such-and-so widget goes at this pixel coordinate, and this other widget goes at that pixel coordinate, and so on?

Many developers have taken that approach — known as absolute positioning – over the years, to their eventual regret.

For example, many of you may have used Windows apps, back in the 1990's, where when you would resize the application window, the app would not really react all that much. You would expand the window, and the UI would not change, except to have big empty areas to the right and bottom of the window. This is because the developers simply said that such-and-so widget goes at this pixel coordinate, and this other widget goes at that pixel coordinate, **regardless of the actual window size**.

In modern Web development, you see this in the debate over fixed versus fluid Web design. The consensus seems to be that fluid designs are better, though frequently they are more difficult to set up. Fluid Web designs can better handle differing browser window sizes, whether those window sizes are because the user resized their browser window manually, or because those window sizes are dictated by the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

screen resolution of the device viewing the Web page. Fixed Web designs — effectively saying that such-and-so element goes at such-and-so pixel coordinate and so on — tend to be easier to build but adapt more poorly to differing browser window sizes.

In mobile, particularly with Android, we have a wide range of possible screen resolutions, from QVGA (320x240) to beyond 1080p (1920x1080), and many values in between. Moreover, any device manufacturer is welcome to create a device with whatever resolution they so desire – there are no rules limiting manufacturers to certain resolutions. Hence, as developers, having the Android equivalent of fluid Web designs is critical, and the way you will accomplish that is by sensible use of containers, avoiding absolute positioning. The containers (and, to a lesser extent, the widgets) will determine how extra space is employed, as the screens get larger and larger.

The Theme of This Section: Themes

In Web development, we have had stylesheets for quite a while. Through such Cascading Style Sheets (CSS) files, we can stipulate various rules about how our Web pages should look. This includes:

- Establishing a default look for certain HTML tags by tag name (e.g., setting the font and size for all `<h1>` and `<h2>` elements)
- Establishing a look for specific HTML elements by class or ID (e.g., setting the width of a specific `<div>` to a certain number of CSS pixels)

In Android, the equivalent concepts can be found in styles and themes. Styles are a collection of values for properties (e.g., have a foreground color of red). These can be applied to specific widgets (e.g., this label should adopt this style), or they can be employed by "themes" that provide the default look for all sorts of widgets and other elements of our UI.

Of course, you do not have to declare any theme for your app. Android will give you a default look-and-feel without any specific theme. That look-and-feel has varied over the years, though, affecting the visual fundamentals of various Android widgets. These themes have names by which we refer to them: `Theme`, `Theme.Holo`, and `Theme.Material`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In the Beginning, There Was "Theme", And It Was Meh

Way back in Android 1.0, the default theme was known simply as `Theme`. Technically, all themes inherit from `Theme`, much as how later CSS stylesheets effectively "inherit" the settings established by prior stylesheets.

The `Theme` UI had a particular look to it:

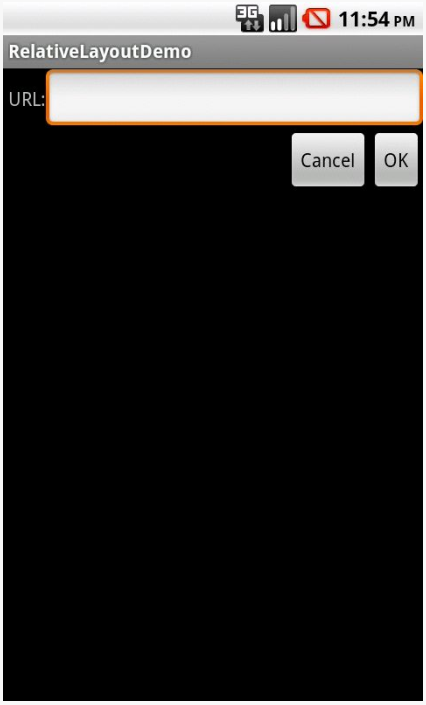


Figure 93: Labels, Fields, and Buttons in Theme

For example:

- At the top of the screen, we had a thin gray "title bar" with the name of our app
- The focused field (an `EditText` widget) had a bright orange outline, whereas normally it was a plain white rectangle
- The buttons ("OK" and "Cancel") were... well... buttons

Holo, There!

Android 3.0 (API Level 11) introduced a new default theme, `Theme.Holo`, with the so-called "holographic widget theme". This changed the look of our UI somewhat:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

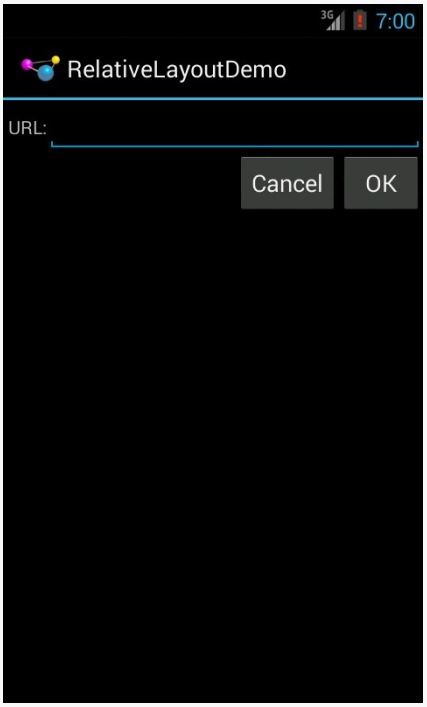


Figure 94: Labels, Fields, and Buttons in Theme.Holo

Now:

- At the top of the screen, we have an "action bar", containing our app's logo and name
- The focused field has a blue "underbracket", whereas normally it is gray
- The buttons are styled slightly differently, with a bigger font, alternative backgrounds, etc.

Considering the Material

Android 5.0 changed the default theme yet again, to `Theme.Material`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

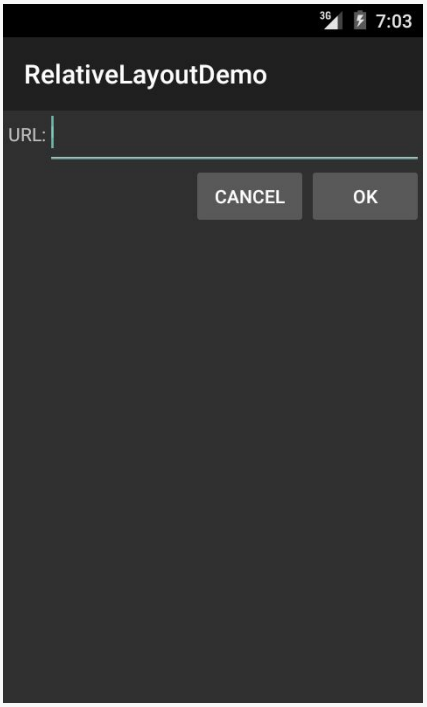


Figure 95: Labels, Fields, and Buttons in Theme.Material

Now:

- The action bar at the top of the screen no longer shows the app icon
- Our field is indicated by an underline, which is teal when focused or gray when unfocused
- The buttons are now forced into all-caps font, with a slightly smaller font size and subtly different background than we had with `Theme.Holo`

Doing More with Themes

Of course, we can do a lot more than just use these. There are other stock themes, with different characteristics. Furthermore, we can customize the themes, by defining our own (inheriting from a stock theme) and changing some of the properties (e.g., replacing the teal color with something else).

We will get much more into creating custom styles and themes later in the book.

However, we will see the effects of `Theme`, `Theme.Holo`, and `Theme.Material` on stock widgets in an upcoming chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Android User Interface

The project you created in an earlier tutorial was just the default files generated by the Android build tools — you did not write any Java code yourself. In this chapter, we will examine the basic Java code and resources that make up an Android activity.

The Activity

The Java source code that you maintain will be in a standard Java-style tree of directories based upon the Java package you chose when you created the project (e.g., `com.commonsware.android` results in `com/commonsware/android/`). Where those files reside depends upon what tools you are using:

- Android Studio will have that source, by default, in `app/src/main/java/` off of the top-level project root
- Eclipse will have that source be in `src/` off of the top-level project root

If you checked the checkbox in your IDE's new-project wizard to create an activity, you will have, in the innermost directory, a Java source file representing an activity class.

A very simple activity looks like:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.os.Bundle;

public class EmPubLiteActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    setContentView(R.layout.main);
  }
}
```

Dissecting the Activity

Let's examine this Java code piece by piece:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.os.Bundle;
```

By default, the package declaration is the same as the one you used when creating the project. And, like any other Java project, you need to import any classes you reference. Most of the Android-specific classes are in the `android` package.

Remember that not every Java SE class is available to Android programs! Visit the Android class reference to see what is and is not available.

```
public class EmPubLiteActivity extends Activity {
```

Activities are public classes, inheriting from the `android.app.Activity` base class (or, possibly, from some other class that itself inherits from `Activity`). You can have whatever data members you decide that you need, though the initial code has none.

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }
```

The `onCreate()` method is invoked when the activity is started. We will discuss the `Bundle` parameter to `onCreate()` in a later chapter. For the moment, consider it an opaque handle that all activities receive upon creation.

The first thing you normally should do in `onCreate()` is chain upward to the superclass, so the stock Android activity initialization can be done. The only other statement in our stub project's `onCreate()` is a call to `setContentView()`. This is where we tell Android what the user interface is supposed to be for our activity.

This raises the question: what does `R.layout.main` mean? Where did this `R` come from?

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To explain that, we need to start thinking about layout resources and how resources are referenced from within Java code.

Using XML-Based Layouts

As noted in the previous chapter, Android uses a series of widgets and containers to describe your typical user interface. These all inherit from an `android.view.View` base class, for things that can be rendered into a standard widget-based activity.

While it is technically possible to create and attach widgets and containers to our activity purely through Java code, the more common approach is to use an XML-based layout file. Dynamic instantiation of widgets is reserved for more complicated scenarios, where the widgets are not known at compile-time (e.g., populating a column of radio buttons based on data retrieved off the Internet).

With that in mind, it's time to break out the XML and learn how to lay out Android activity contents that way.

What Is an XML-Based Layout?

As the name suggests, an XML-based layout is a specification of its widgets' relationships to each other — and to containers — encoded in XML format. Specifically, Android considers XML-based layouts to be resources, and as such layout files are stored in the `res/layout/` directory inside your Android project (or, as we will see later, other layout resource sets, like `res/layout-land/` for layouts to use when the device is held in landscape). As has been noted elsewhere in this book, the location of `res/` is either in the project root for Eclipse or in `app/src/main/` for Android Studio.

Each XML file contains a tree of elements specifying a layout of widgets and containers that make up one `View`. The attributes of the XML elements are properties, describing how a widget should look or how a container should behave. For example, if a `Button` element has an attribute value of `android:textStyle = "bold"`, that means that the text appearing on the face of the button should be rendered in a boldface font style.

For example, here is a `res/layout/main.xml` file that could be used with the aforementioned activity:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".EmPubLiteActivity">

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:text="@string/hello_world" />

</RelativeLayout>
```

The class name of a widget or container — such as `RelativeLayout` or `TextView` – forms the name of the XML element. Since `TextView` is an Android-supplied widget, we can just use the bare class name. If you create your own widgets as subclasses of `android.view.View`, you would need to provide a full package declaration as well (e.g., `com.commonsware.android.MyWidget`).

The root element needs to declare the Android XML namespace (`xmlns:android="http://schemas.android.com/apk/res/android"`). All other elements will be children of the root and will inherit that namespace declaration.

The attributes are properties of the widget or container, describing what it should look and work like. For example, the `android:layout_centerHorizontal="true"` attribute on the `TextView` element indicates that the `TextView` should be centered within its `RelativeLayout` parent.

We will get into details about these attributes, their possible values, and their uses, in upcoming chapters. Note that those attributes in the `tools` namespace (e.g., `tools:context`) are there solely to support the Android build tools, and do not affect the runtime execution of your project.

Android's SDK ships with a tool (`aapt`) which uses the layouts. This tool will be automatically invoked by your Android tool chain (e.g., Android Studio, Eclipse). Of particular importance to you as a developer is that `aapt` generates an `R.java` source file, allowing you to access layouts and widgets within those layouts directly from your Java code. In other words, this is where that magic `R` value used in `setContentView()` comes from. We will discuss that a bit more later in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

XML Layouts and Your IDE

If you are using Android Studio or Eclipse, and you double-click on the `res/layout/main.xml` file in your project, you will not initially see that XML. Instead, you will be taken to the graphical layout editor:

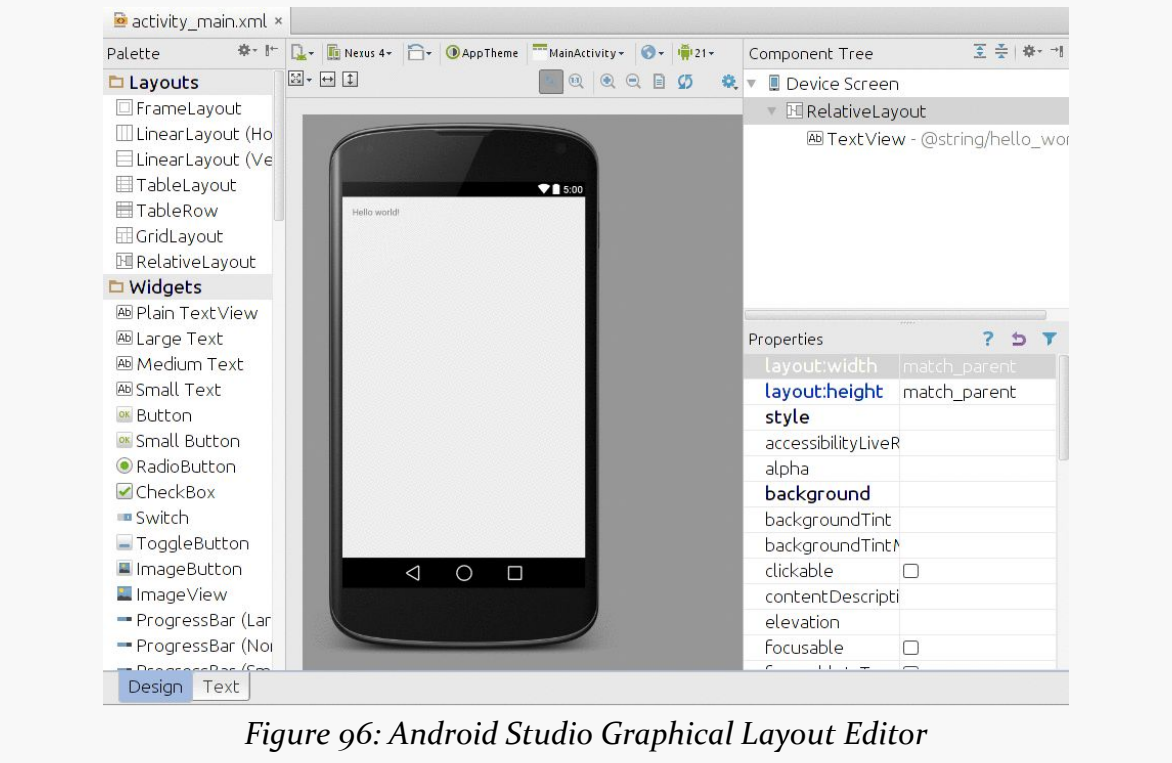


Figure 96: Android Studio Graphical Layout Editor

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

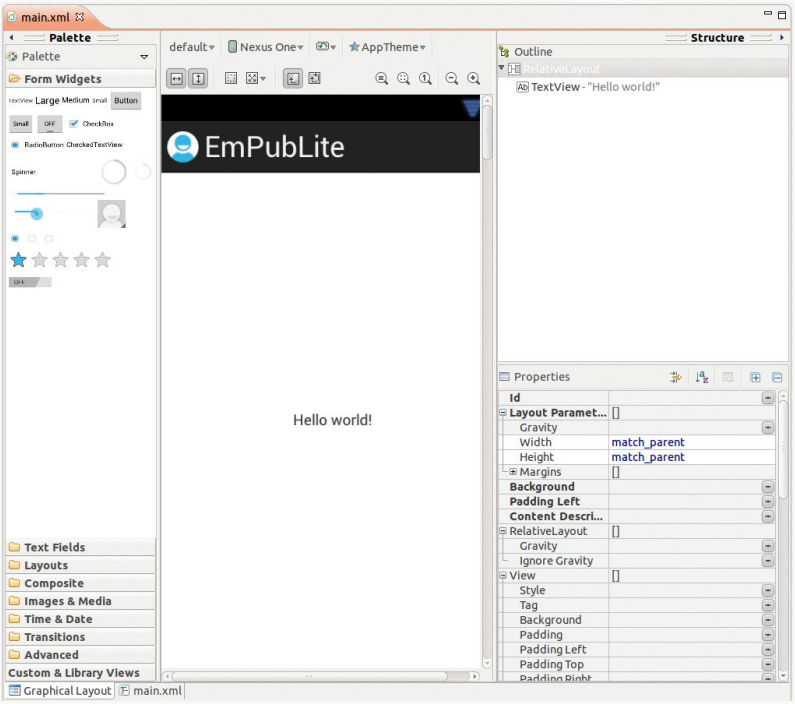


Figure 97: Eclipse Graphical Layout Editor

A sub-tab (e.g., "Text" on Android Studio) will show you the raw XML. The default "Design" or "Graphical Layout" sub-tab, though, shows you a preview of what your layout would look like, if it were to be used for an activity. The "Palette" on the left shows all sorts of widgets and containers, which you can drag into the preview area to add an instance of your chosen widget or container to your layout. Right-clicking over a widget or container will give you an extensive context menu to configure the item, and the toolbar immediately above the preview area will let you configure common properties of a selected widget or container.

We will go into much more detail about using the graphical layout editor in an upcoming chapter, as we start to work more with specific widgets and containers.

Why Use XML-Based Layouts?

Almost everything you do using XML layout files can be achieved through Java code. For example, you could use `setText()` to have a button display a certain caption, instead of using a property in an XML layout. Since XML layouts are yet another file for you to keep track of, we need good reasons for using such files.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Perhaps the biggest reason is to assist in the creation of tools for view definition, such as the aforementioned graphical layout editors in Android Studio and Eclipse. Such GUI builders could, in principle, generate Java code instead of XML. The challenge is re-reading the definition in to support edits — that is far simpler if the data is in a structured format like XML than in a programming language. Moreover, keeping the generated bits separated out from hand-written code makes it less likely that somebody's custom-crafted source will get clobbered by accident when the generated bits get re-generated. XML forms a nice middle ground between something that is easy for tool-writers to use and easy for programmers to work with by hand as needed.

Also, XML as a GUI definition format is becoming more commonplace. Microsoft's XAML, Adobe's Flex, Google's GWT, and Mozilla's XUL all take a similar approach to that of Android: put layout details in an XML file and put programming smarts in source files (e.g., JavaScript for XUL). Many less-well-known GUI frameworks, such as ZK, also use XML for view definition. While "following the herd" is not necessarily the best policy, it does have the advantage of helping to ease the transition into Android from any other XML-centered view description language.

Using Layouts from Java

Given that you have painstakingly set up the widgets and containers for your view in an XML layout file named `main.xml` stored in `res/layout/`, all you need is one statement in your activity's `onCreate()` callback to use that layout, as we saw in our stub project's activity:

```
setContentView(R.layout.main);
```

Here, `R.layout.main` tells Android to load in the layout (`layout`) resource (`R`) named main.xml (`main`).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Basic Widgets

Every GUI toolkit has some basic widgets: fields, labels, buttons, etc. Android's toolkit is no different in scope, and the basic widgets will provide a good introduction as to how widgets work in Android activities. We will examine a number of these in this chapter.

Common Concepts

There are a few core features of widgets that we need to discuss at the outset, before we dive into details on specific types of widgets.

Widgets and Attributes

As mentioned in a previous chapter, widgets have attributes that describe how they should behave. In an XML layout file, these are literally XML attributes on the widget's element in the file. Usually, there are corresponding getter and setter methods for manipulating this attribute at runtime from your Java code.

If you visit the JavaDocs for a widget, such as the JavaDocs for `TextView`, you will see an "XML Attributes" table near the top. This lists all of the attributes defined uniquely on this class, and the "Inherited XML Attributes" table that follows lists all those that the widget inherits from superclasses, such as `View`. Of course, the JavaDocs also list the fields, constants, constructors, and public/protected methods that you can use on the widget itself.

Those attributes can be modified by a "Properties" view in the graphical layout editor of the IDE:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

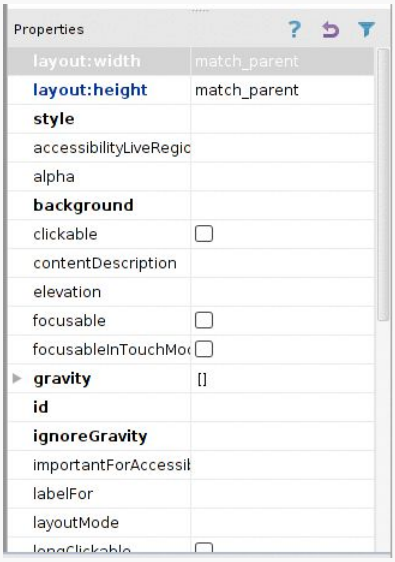


Figure 98: Properties View in Android Studio Graphical Layout Editor

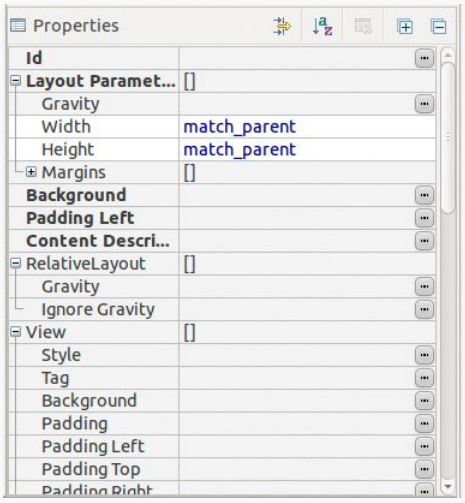


Figure 99: Properties View in Eclipse Graphical Layout Editor

This book does not attempt to explain each and every attribute on each and every widget. We will, however, cover the most popular widgets and the most commonly-used attributes on those widgets.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Referencing Widgets By ID

Many widgets and containers only need to appear in the XML layout file and do not need to be referenced in your Java code. For example, a static label (`TextView`) frequently only needs to be in the layout file to indicate where it should appear.

Anything you *do* want to use in your Java source, though, needs an `android:id`.

The convention is to use `@+id/...` as the `id` value, where the `...` represents your locally-unique name for the widget in question, for the first occurrence of a given `id` value in your layout file. The second and subsequent occurrences in the same layout file should drop the `+` sign.

Android provides a few special `android:id` values, of the form `@android:id/...` — we will see some of these in various chapters of this book.

To access our identified widgets, use `findViewById()`, passing it the numeric identifier of the widget in question. That numeric identifier was generated by Android in the `R` class as `R.id.something` (where `something` is the specific widget you are seeking).

This concept will become important as we try to attach listeners to our widgets (e.g., finding out when a checkbox is checked) or when we try referencing widgets from other widgets in a layout XML file (e.g., with `RelativeLayout`). All of this will be covered later in this chapter.

Size

Most of the time, we need to tell Android how big we want our widgets to be. Occasionally, this will be handled for us — we will see an example of that with `TableLayout` in an upcoming chapter. But generally we need to provide this information ourselves.

To do that, you need to supply `android:layout_width` and `android:layout_height` attributes on your widgets in the XML layout file. These attributes' values have three flavors:

1. You can provide a specific dimension, such as `125dip` to indicate the widget should take up exactly a certain size (here, 125 density-independent pixels)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. You can provide `wrap_content`, which means the widget should take up as much room as its contents require (e.g., a `TextView` label widget's content is the text to be displayed)
3. You can provide `match_parent`, which means the widget should fill up all remaining available space in its enclosing container

The latter two flavors are the most common, as they are independent of screen size, allowing Android to adjust your view to fit the available space.

Note that you will also see `fill_parent`. This is an older synonym for `match_parent`. `match_parent` is the recommended value going forward, but `fill_parent` will certainly work.

This chapter focuses on individual widgets. Size becomes much more important when we start combining multiple widgets on the screen at once, and so we will be spending more time on sizing scenarios in later chapters.

The `layout_` prefix on these attributes means that these attributes represent requests by the widget to its enclosing container. Whether those requests will be truly honored will depend a bit on what other widgets there are in the container and what *their* requests are.

Assigning Labels

The simplest widget is the label, referred to in Android as a `TextView`. Like in most GUI toolkits, labels are bits of text not editable directly by users. Typically, they are used to identify adjacent widgets (e.g., a "Name:" label before a field where one fills in a name).

In Java, you can create a label by creating a `TextView` instance. More commonly, though, you will create labels in XML layout files by adding a `TextView` element to the layout, with an `android:text` property to set the value of the label itself. If you need to swap labels based on certain criteria, such as internationalization, you may wish to use a string resource reference in `android:text` instead (e.g., `@string/label`).

For example, in our last tutorial, we still are using the automatically-generated `res/layout/main.xml` file, containing, among other things, a `TextView`:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".EmPubLiteActivity">

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:text="@string/hello_world"/>

</RelativeLayout>
```

Android Studio Graphical Layout Editor

The `TextView` widget is available in the "Widgets" category of the Palette in the Android Studio graphical layout editor:

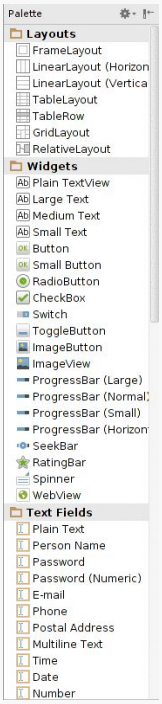


Figure 100: Palette, "Plain TextView" in Widgets Category

The "Large Text", "Medium Text", and "Small Text" items beneath "Plain TextView" are also `TextView` widgets, with different default font sizes.

You can drag that `TextView` from the palette into a layout file in the main editing area to add the widget to the layout. Or, drag it over the top of some container you

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

see in the Component Tree pane of the editor to add it as a child of that specific container:

Figure 101: Component Tree Pane

Clicking on the resulting `TextView` in the Component Tree pane, or in the preview area, will set up the Properties pane with the various attributes of the widget, ready for you to change as needed:

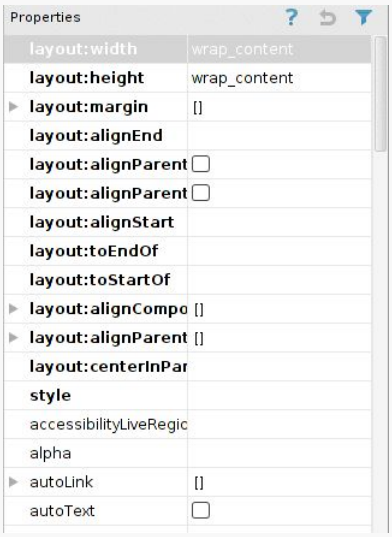


Figure 102: Properties Pane, for a TextView Inside a RelativeLayout

Editing the Text

The “Text” property will allow you to choose or define a string resource to serve as the text to be displayed:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

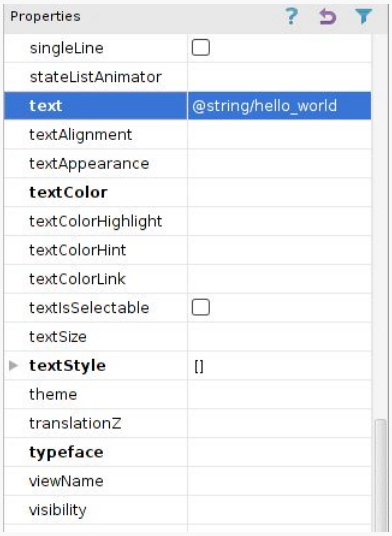


Figure 103: Properties Pane, with TextView "text" Property Selected

Clicking on the value will allow the property to be edited:

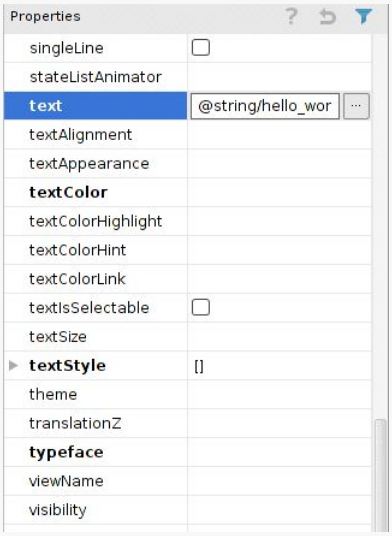


Figure 104: Properties Pane, with TextView "text" Property Editable

You can either type a literal string right in the Properties pane row, or you can click the "..." button to the right of the field to pick a string resource:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

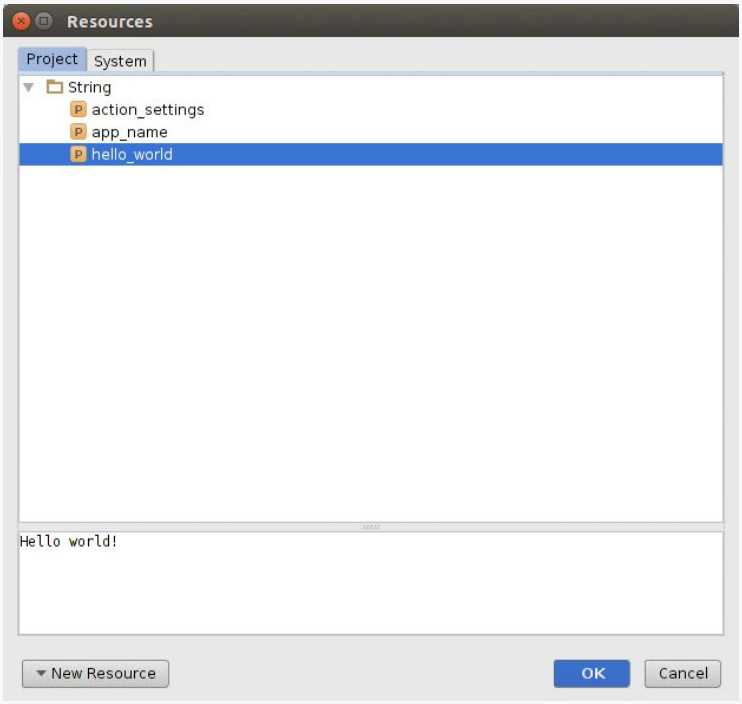


Figure 105: String Resources Dialog

You can highlight one of those resources and click "OK" to use it. Or, in the bottom of that dialog, there is a "New Resource" drop-down. When viewing string resources, that drop-down will contain a single command: "New String Value...". Choosing it will allow you to define a new string resource via another dialog:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

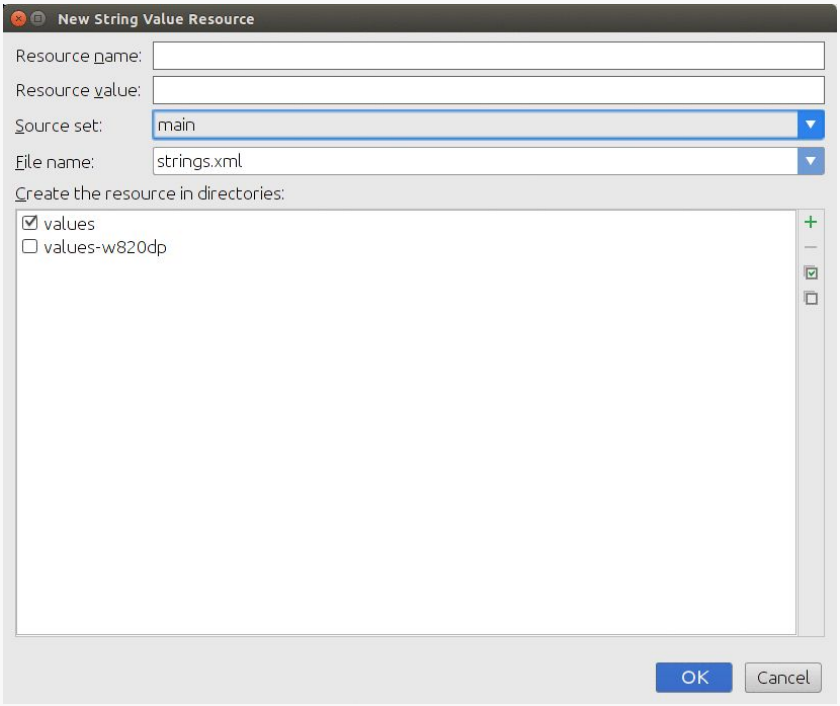


Figure 106: New String Resource Dialog

You can give your new string resource a name, the actual text of the string itself, the filename in which the string resource should reside (`strings.xml` by default), and which `values/` directory the string should go into (`values` by default). You will also choose the "source set" — for now, that will just be `main`. Once you accept the dialog, your new string resource will be applied to your `TextView`.

Editing the ID

The "id" property will allow you to change the `android:id` value of the widget:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

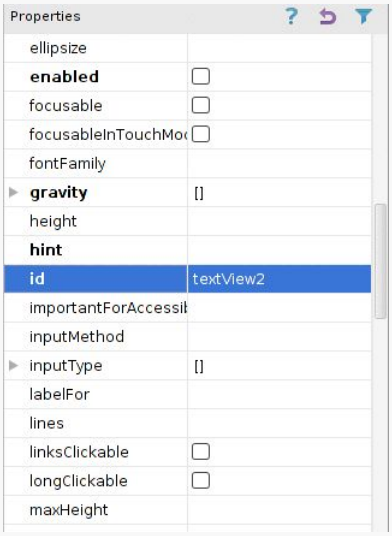


Figure 107: Properties Pane, with TextView "id" Property Selected

The value you fill in here is what goes after the `@+id/` portion (e.g., `textView2`).

Eclipse Graphical Layout Editor

The `TextView` widget is available in the "Form Widgets" portion of the Palette in the Eclipse graphical layout editor:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

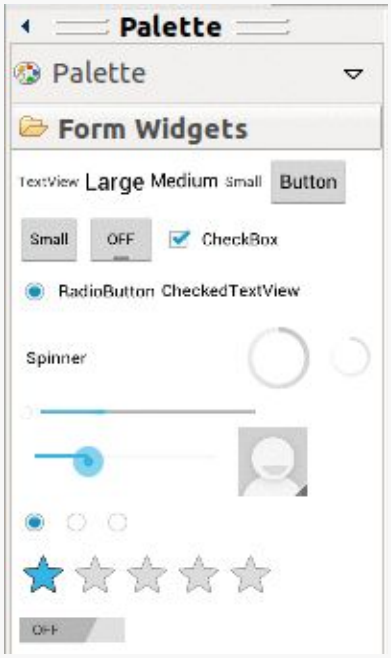


Figure 108: Form Widgets Palette, TextView in Upper Left

You can drag that `TextView` from the palette into a layout file in the main editing area to add the widget to the layout. Or, drag it over the top of some container you see in the Outline pane of the editor to add it as a child of that specific container:

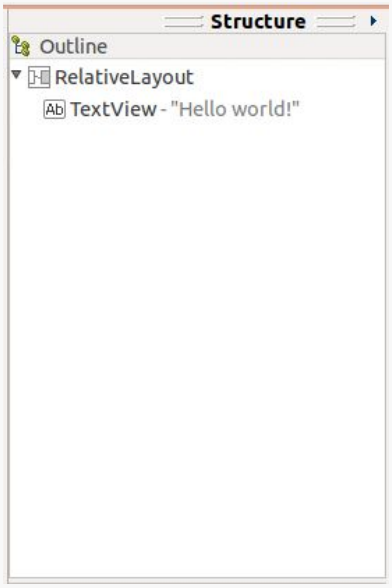


Figure 109: Outline Pane

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Clicking on the resulting `TextView` in the Outline pane, or in the preview area, will set up the Properties pane with the various attributes of the widget, ready for you to change as needed:

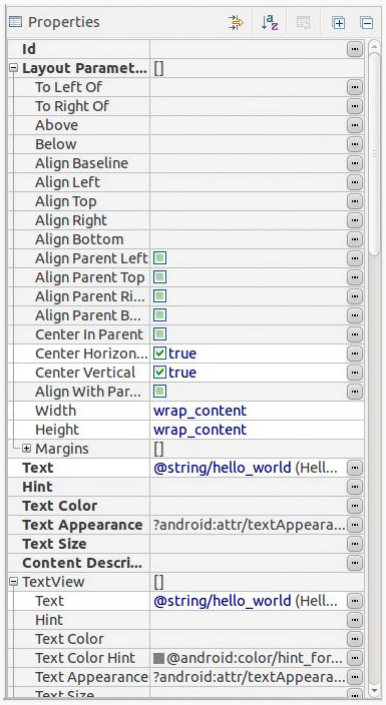


Figure 110: Properties Pane, for a TextView Inside a RelativeLayout

Editing the Text

The "Text" property will allow you to choose or define a string resource to serve as the text to be displayed. By default, it brings up a list of existing string resources:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

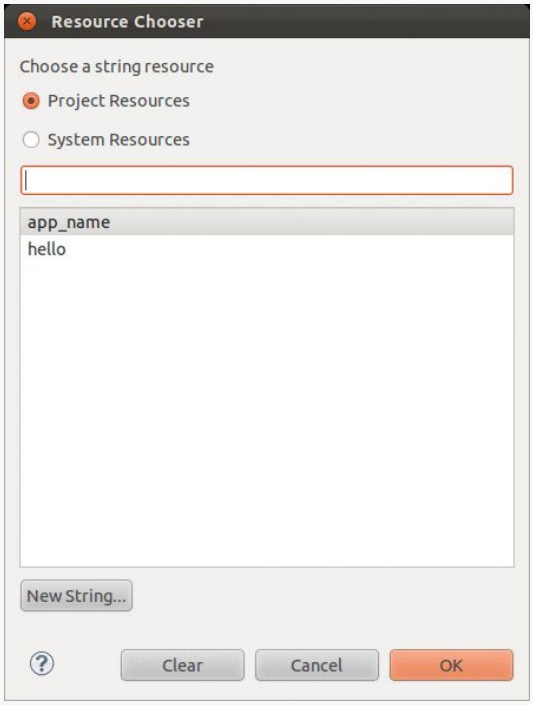


Figure 111: String Resource Chooser

You can highlight one of those resources and click "OK" to use it, or you can click the "New String..." button to define a brand-new string resource.

Editing the ID

The "Id" property will allow you to change the `android:id` value of the widget. Be sure to include the `@+id/` prefix, as Android will not add that automatically for you.

Notable TextView Attributes

`TextView` has numerous other attributes of relevance for labels, such as:

1. `android:typeface` to set the typeface to use for the label (e.g., `monospace`)
2. `android:textStyle` to indicate that the typeface should be made bold (`bold`), italic (`italic`), or bold and italic (`bold_italic`)
3. `android:textColor` to set the color of the label's text, in RGB hex format (e.g., `#FF0000` for red) or ARGB hex format (e.g., `#88FF0000` for a translucent red)

These attributes, like most others, can be modified through the Properties pane.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For example, in the Basic/Label sample project, you will find the following layout file:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="@string/profound"
  />
```

Just that layout alone, with the stub Java source provided by Android's project builder (e.g., `android create project`) and appropriate string resources, gives you:

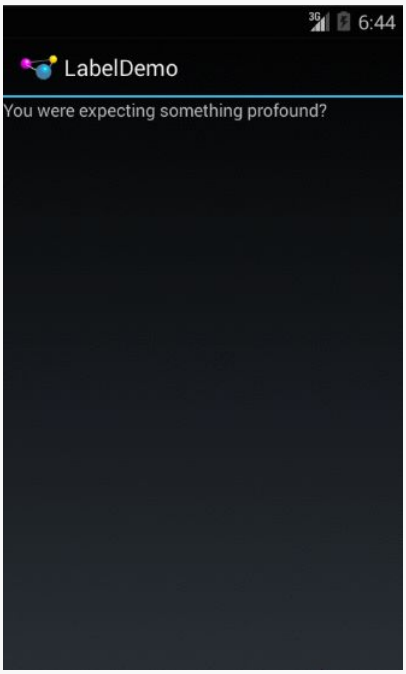


Figure 112: The LabelDemo Sample Application

A Commanding Button

Android has a `Button` widget, which is your classic push-button "click me and something cool will happen" widget. As it turns out, `Button` is a subclass of `TextView`, so everything discussed in the preceding section in terms of formatting the face of the button still holds.

For example, in the Basic/Button sample project, you will find the following layout file:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button"/>

</LinearLayout>
```

Just that layout alone, with the stub Java source provided by Android's project builder (e.g., `android create project`) and appropriate string resources, gives you:

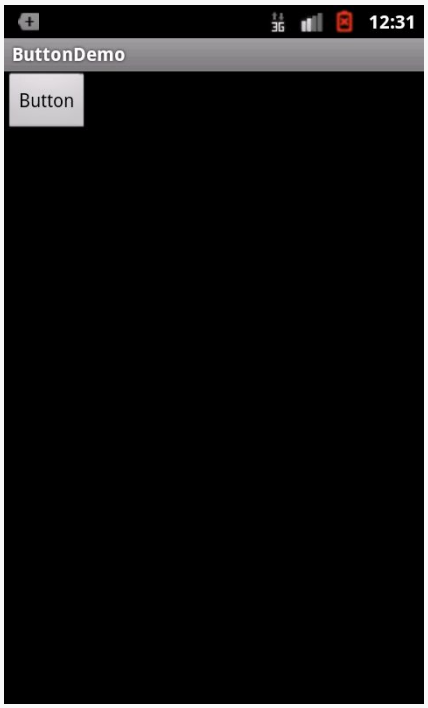


Figure 113: Button Widget, in Theme

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 114: Button Widget, in Theme.Holo

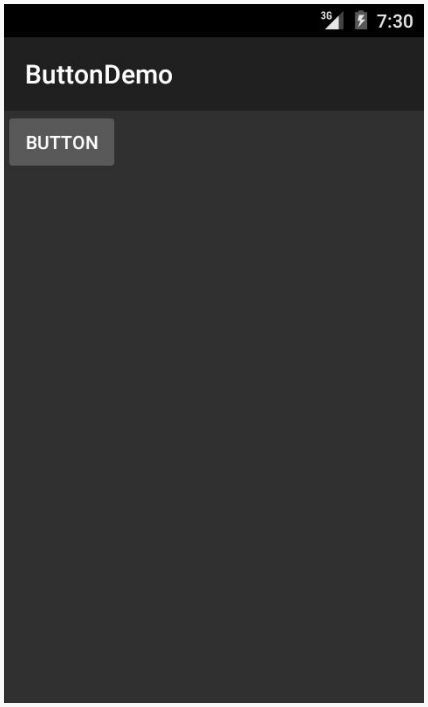


Figure 115: Button Widget, in Theme.Material

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio Graphical Layout Editor

As with the `TextView` widget, the `Button` widget is available in the "Widgets" portion of the Palette in the Android Studio graphical layout editor:

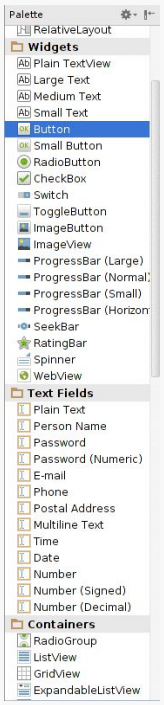


Figure 116: Widgets Palette, Button Shown Highlighted

You can drag that `Button` from the palette into a layout file in the main editing area to add the widget to the layout. The Properties pane will then let you adjust the various attributes of this `Button`. Since `Button` inherits from `TextView`, most of the options are the same (e.g., "Text").

Eclipse Graphical Layout Editor

As with the `TextView` widget, the `Button` widget is available in the "Form Widgets" portion of the Palette in the Eclipse graphical layout editor:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

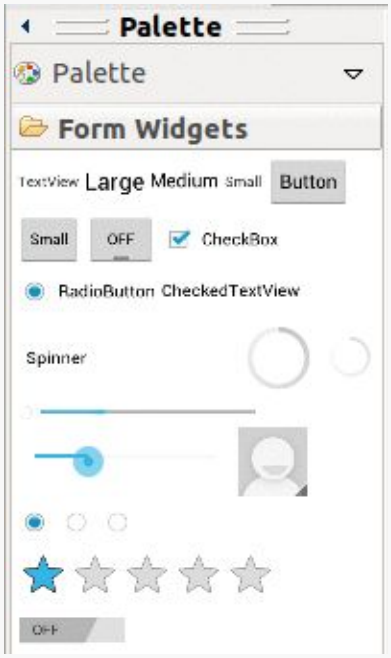


Figure 117: Form Widgets Palette, Button in Upper Right

You can drag that `Button` from the palette into a layout file in the main editing area to add the widget to the layout. The Properties pane will then let you adjust the various attributes of this `Button`. Since `Button` inherits from `TextView`, most of the options are the same (e.g., "Text").

Tracking Button Clicks

Buttons are command widgets — when the user presses a button, they expect something to happen.

To define what happens when you click a `Button`, you can:

1. Define some method on your `Activity` that holds the button that takes a single `View` parameter, has a `void` return value, and is `public`
2. In your layout XML, on the `Button` element, include the `android:onClick` attribute with the name of the method you defined in the previous step

For example, we might have a method on our `Activity` that looks like:

```
public void someMethod(View theButton) {
  // do something useful here
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Then, we could use this XML declaration for the `Button` itself, including `android:onClick`:

```
<Button
  android:onClick="someMethod"
  ...
/>
```

This is enough for Android to "wire together" the `Button` with the click handler. When the user clicks the button, `someMethod()` will be called.

Another approach is to skip `android:onClick`, instead calling `setOnClickListener()` on the `Button` object in Java code. When a `Button` is used directly by an activity, this is not typically used — `android:onClick` is a bit cleaner. However, when we start to talk about fragments, you will see that `android:onClick` does not work that well with fragments, and so we will use `setOnClickListener()` at that point.

Fleeting Images

Android has two widgets to help you embed images in your activities: `ImageView` and `ImageButton`. As the names suggest, they are image-based analogues to `TextView` and `Button`, respectively.

Each widget takes an `android:src` attribute (in an XML layout) to specify what picture to use. These usually reference a drawable resource (e.g., `@drawable/icon`).

`ImageButton`, a subclass of `ImageView`, mixes in the standard `Button` behaviors, for responding to clicks and whatnot.

For example, take a peek at the `main.xml` layout from the `Basic/ImageView` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<ImageView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/icon"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:adjustViewBounds="true"
  android:src="@drawable/molecule"/>
```

The result, just using the code-generated activity, is simply the image:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

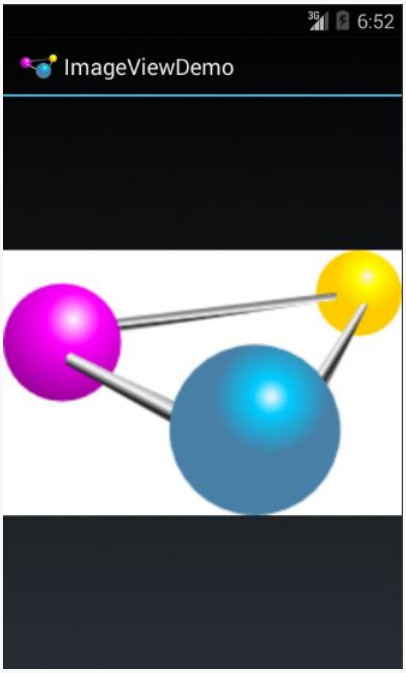


Figure 118: The ImageViewDemo sample application

Android Studio Graphical Layout Editor

The `ImageView` widget can be found in the "Widgets" portion of the Palette in the Android Studio Graphical Layout editor:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

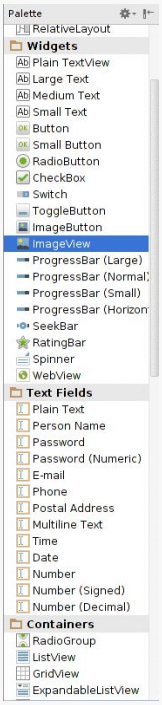


Figure 119: Widgets Palette, ImageView Shown Highlighted

`ImageButton` appears alongside `ImageView` in that tool palette.

You can drag these into a layout file, then use the Properties pane to set their attributes. Like all widgets, you will have an "id" option to set the `android:id` value for the widget. Two others of importance, though, are more unique to `ImageView` and `ImageButton`:

- "src" allows you to choose a drawable resource to use as the image to be displayed
- "scaleType" opens a drop-down menu where you can choose how the image is to be scaled:

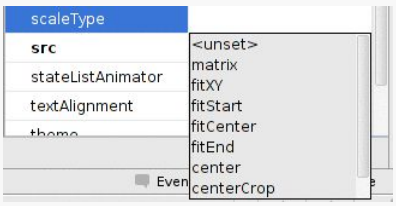


Figure 120: Scale Types in Android Studio Properties Pane

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We will examine those scale types more later in this chapter.

Eclipse Graphical Layout Editor

The `ImageView` widget can be found in the "Images & Media" portion of the Palette in the Graphical Layout editor:

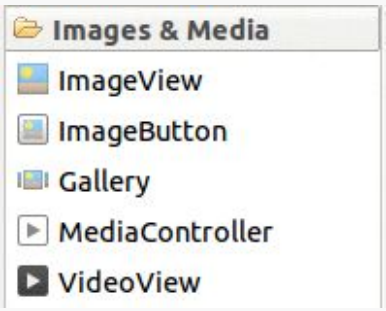


Figure 121: Images & Media Widgets Palette, ImageView in Upper Left

The `ImageButton` widget is adjacent to the `ImageView` widget in the Palette.

You can drag these into a layout file, then use the Properties pane to set their attributes. Like all widgets, you will have an "Id" option to set the `android:id` value for the widget. Two others of importance, though, are more unique to `ImageView` and `ImageButton`:

- "Src" allows you to choose a drawable resource to use as the image to be displayed
- "Scale Type" opens a drop-down menu where you can choose how the image is to be scaled:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

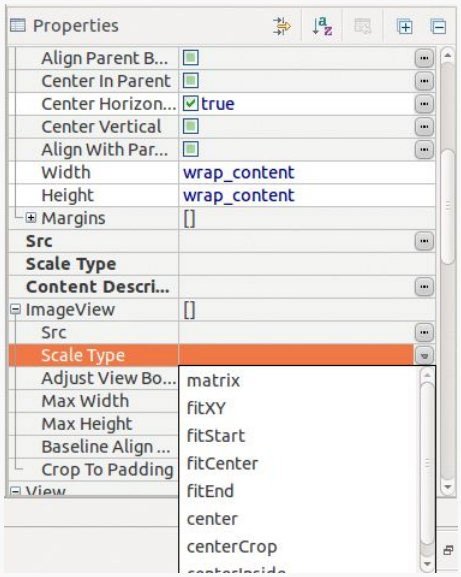


Figure 122: Scale Types in Eclipse Properties Pane

We will examine those scale types more in the next section.

Scaling Images

It is possible, perhaps even probable, that our `ImageView` size will not exactly match the size of the images that we are trying to display. `ImageView` supports a variety of "scale types" that indicate how Android should try to deal with the discrepancy between the size/aspect ratio of the image and the size/aspect ratio of the `ImageView` itself.

These values can be seen in the JavaDocs in the `ImageView.ScaleType` class. The default ("fitCenter") simply scales up the image to best fit the available space.

Of note, a choice of "center" will center the image in the available space but will not scale up the image:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

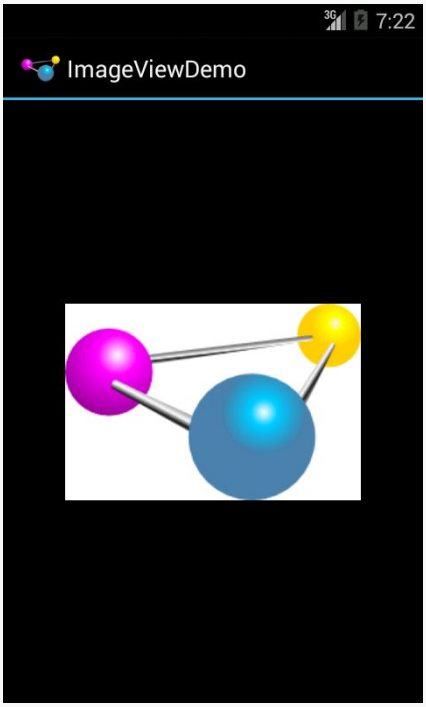


Figure 123: The ImageViewDemo Sample, Set to center

A choice of "centerCrop" will scale the image so that its shortest dimension fills the available space and crops the rest:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

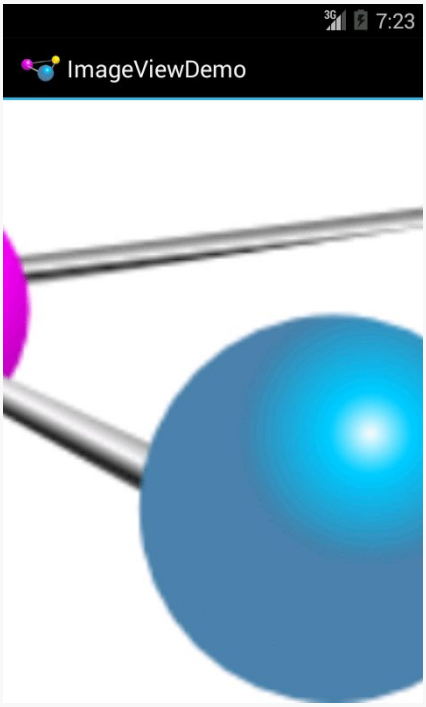


Figure 124: The ImageViewDemo Sample, Set to centerCrop

A choice of "fitXY" will scale the image to fill the space, ignoring the aspect ratio:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

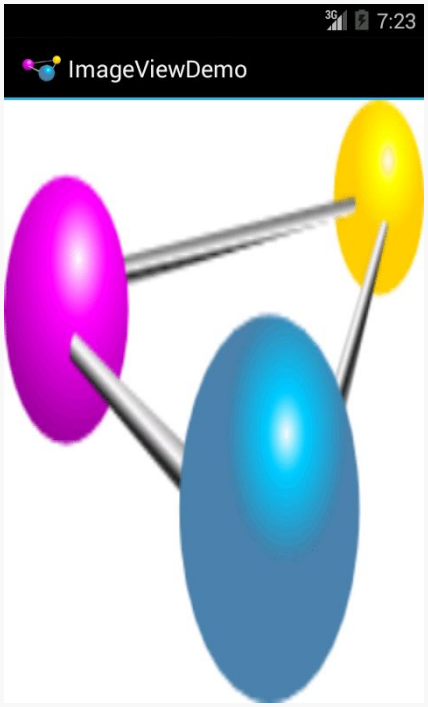


Figure 125: The ImageViewDemo Sample, Set to fitXY

Fields of Green. Or Other Colors.

Along with buttons and labels, fields are the third "anchor" of most GUI toolkits. In Android, they are implemented via the `EditText` widget, which is a subclass of the `TextView` used for labels.

Along with the standard `TextView` attributes (e.g., `android:textStyle`), `EditText` has others that will be useful for you in constructing fields, notably `android:inputType`, to describe what sort of input your `EditText` expects (numbers? email addresses? phone numbers?). A thorough explanation of `android:inputType` and its interaction with input method editors (a.k.a., "soft keyboards") will be discussed in an upcoming chapter.

For example, from the `Basic/Field` sample project, here is an XML layout file showing an `EditText`:

```
<?xml version="1.0" encoding="utf-8"?>
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/field"
  android:layout_width="match_parent"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:layout_height="match_parent"
  android:inputType="textMultiLine"
  android:text="@string/license"
  />
```

Note that we have `android:inputType="textMultiLine"`, so users will be able to enter in several lines of text. We also have defined the initial text to be the value of a `license` string resource.

The result, once built and installed into the emulator, is:

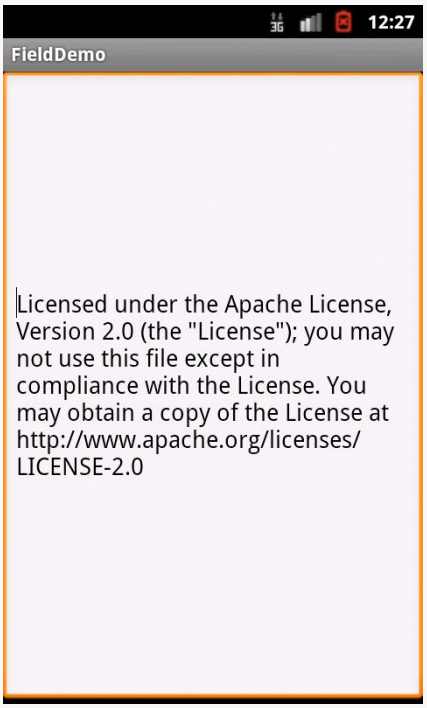


Figure 126: FieldDemo, in Theme

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

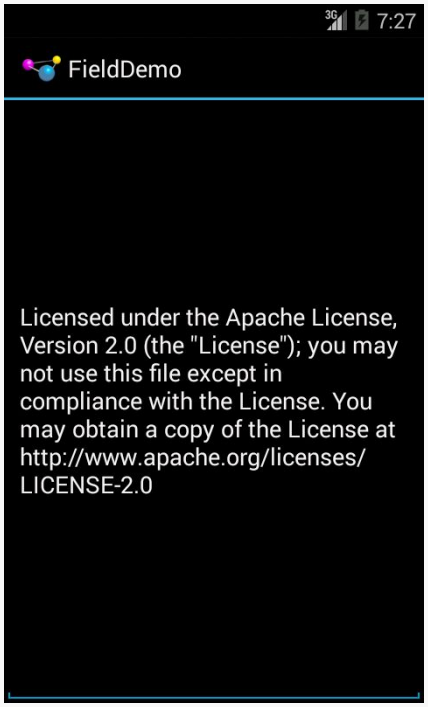


Figure 127: FieldDemo, in Theme.Holo

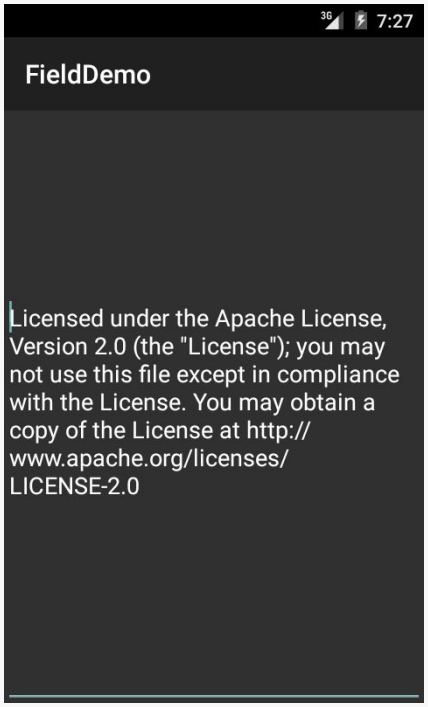


Figure 128: FieldDemo, in Theme.Material

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio Graphical Layout Editor

The Android Studio Graphical Layout's Palette has a whole section dedicated primarily to `EditText` widgets, named "Text Fields":

Figure 129: Text Fields Palette

The first entry is a general-purpose `EditText`. The rest come pre-configured for various scenarios, such as a person's name or a password.

You can drag any of these into your layout, then use the Properties pane to configure relevant attributes. The "Id" and "Text" attributes are the same as found on `TextView`, as are many other properties, as `EditText` inherits from `TextView`.

Eclipse Graphical Layout Editor

The Graphical Layout's Palette has a whole section dedicated primarily to `EditText` widgets, named "Text Fields":

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

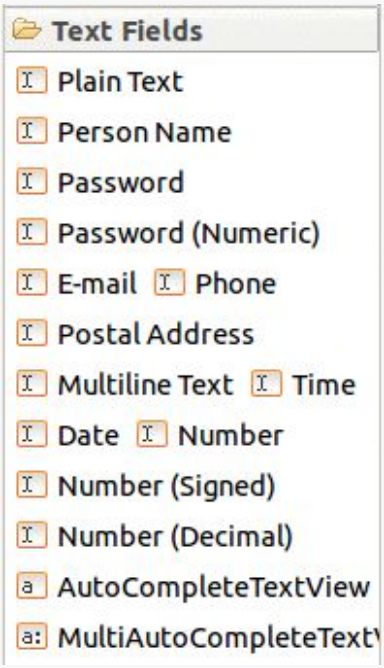


Figure 130: Text Fields Palette

The first entry is a general-purpose `EditText`. The rest come pre-configured for various scenarios, such as a person's name or a postal address.

You can drag any of these into your layout, then use the Properties pane to configure relevant attributes. The "Id" and "Text" attributes are the same as found on `TextView`, as are many other properties, as `EditText` inherits from `TextView`.

Notable EditText Attributes

The "Hint" item in the Properties pane allows you to set a "hint" for this `EditText`. The "hint" text will be shown in light gray in the `EditText` widget when the user has not entered anything yet. Once the user starts typing into the `EditText`, the "hint" vanishes. This might allow you to save on screen space, replacing a separate label `TextView`.

The "Input Type" item in the Properties pane allows you to describe what sort of input you are expecting to receive in this `EditText`, lining up with many of the types of fields you can drag from the Palette into the layout:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

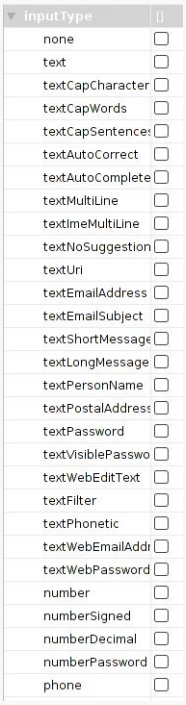


Figure 131: Android Studio's Text Fields InputType Drop-Down

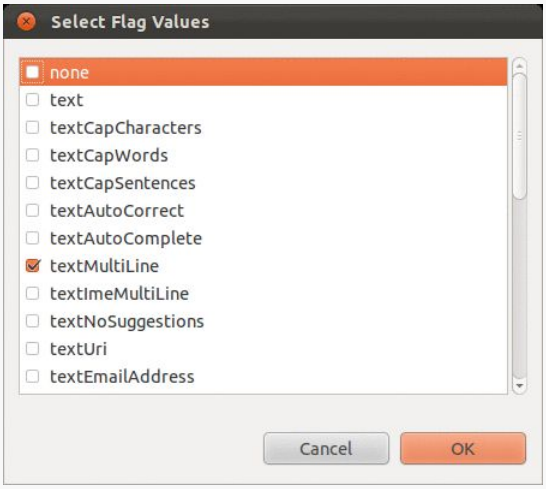


Figure 132: Eclipse's Text Fields InputType Dialog

The `inputType` attribute will be covered in greater detail in an upcoming chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

More Common Concepts

All widgets, including the ones shown above, extend `View`. The `View` base class gives all widgets an array of useful attributes and methods beyond those already described.

Padding

Widgets have a minimum size, one that may be influenced by what is inside of them. So, for example, a `Button` will expand to accommodate the size of its caption. You can control this size using padding. Adding padding will increase the space between the contents (e.g., the caption of a `Button`) and the edges of the widget.

Padding can be set once in XML for all four sides (`android:padding`) or on a per-side basis (`android:paddingLeft`, etc.). Padding can also be set in Java via the `setPadding()` method.

The value of any of these is a dimension — a combination of a unit of measure and a count. So, `10dip` is 10 density-independent pixels, `2mm` is 2 millimeters, etc.

Margins

By default, widgets are tightly packed, one next to the other. You can control this via the use of margins, a concept that is reminiscent of the padding described previously.

The difference between padding and margins comes in terms of the background. For widgets with a transparent background — like the default look of a `TextView` — padding and margins have similar visual effect, increasing the space between the widget and adjacent widgets. However, for widgets with a non-transparent background — like a `Button` — padding is considered inside the background while margins are outside. In other words, adding padding will increase the space between the contents (e.g., the caption of a `Button`) and the edges, while adding margin increases the empty space between the edges and adjacent widgets.

Margins can be set in XML, either on a per-side basis (e.g., `android:layout_marginTop`) or on all sides via `android:layout_margin`. Once again, the value of any of these is a dimension — a combination of a unit of measure and a count, such as `5dp` for 5 density-independent pixels.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Colors

There are two types of color attributes in Android widgets. Some, like `android:background`, take a single color (or a drawable to serve as the background). Others, like `android:textColor` on `TextView` (and subclasses) can take a `ColorStateList`, including via the Java setter (in this case, `setTextColor()`).

A `ColorStateList` allows you to specify different colors for different conditions. For example, when you get to selection widgets in an upcoming chapter, you will see how a `TextView` has a different text color when it is the selected item in a list compared to when it is in the list but not selected. This is handled via the default `ColorStateList` associated with `TextView`.

If you wish to change the color of a `TextView` widget in Java code, you have two main choices:

- Use `ColorStateList.valueOf()`, which returns a `ColorStateList` in which all states are considered to have the same color, which you supply as the parameter to the `valueOf()` method. This is the Java equivalent of the `android:textColor` approach, to make the `TextView` always be a specific color regardless of circumstances.
- Create a `ColorStateList` with different values for different states, either via the constructor or via an XML drawable resource. This will be covered much later in the book.

Other Useful Attributes

Some additional attributes on `View` most likely to be used include:

1. `android:visibility`, which controls whether the widget is initially visible
2. `android:nextFocusDown`, `android:nextFocusLeft`, `android:nextFocusRight`, and `android:nextFocusUp`, which control the focus order if the user uses the D-pad, trackball, or similar pointing device
3. `android:contentDescription`, which is roughly equivalent to the `alt` attribute on an HTML `<img>` tag, and is used by accessibility tools to help people who cannot see the screen navigate the application — this is very important for widgets like `ImageView`

We will see more about the focus attributes and `android:contentDescription` in the chapter on focus management and accessibility, later in this book.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Useful Methods

You can toggle whether or not a widget is enabled via `setEnabled()` and see if it is enabled via `isEnabled()`. One common use pattern for this is to disable some widgets based on a `CheckBox` or `RadioButton` checked state. We will explore `CheckBox`, `RadioButton`, and similar sorts of widgets a bit later in the book.

You can give a widget focus via `requestFocus()` and see if it is focused via `isFocused()`. You might use this in concert with disabling widgets as mentioned above, to ensure the proper widget has the focus once your disabling operation is complete.

To help navigate the tree of widgets and containers that make up an activity's overall view, you can use:

1. `getParent()` to find the parent widget or container
2. `findViewById()` to find a child widget with a certain ID
3. `getRootView()` to get the root of the tree (e.g., what you provided to the activity via `setContentView()`)

Visit the Trails!

You can learn more about Android's input method framework — what you might think of as soft keyboards — in a later chapter.

Another chapter in the trails covers the use of fonts, to tailor your `TextView` widgets (and those that inherit from them, like `Button`).

Yet another chapter in the trails covers rich text formatting, both for presenting formatted text in a `TextView` (e.g., inline **boldface**) and for collecting formatted text from the user via a customized `EditText`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Debugging Your App

Now that we are starting to manipulate layouts and Java code more significantly, the odds increase that we are going to somehow do it wrong, and our app will crash.

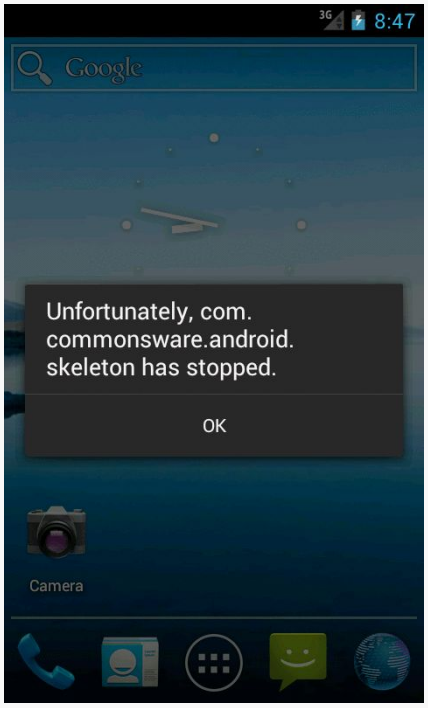


Figure 133: A Crash Dialog on Android 4.0.3

In this chapter, we will cover a few tips on how to debug these sorts of issues.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Get Thee To a Stack Trace

If you see one of those "Force Close" or "Has Stopped" dialogs, the first thing you will want to do is examine the Java stack trace that is associated with this crash. These are logged to a facility known as LogCat, on your device or emulator.

To view LogCat, you have three choices:

1. Use the `adb logcat` command at the command line (or something that uses `adb logcat`, such as various colorizing scripts available online)
2. Use the LogCat tab in the standalone Android Device Monitor utility (run `monitor` from the command line)
3. Use the LogCat view in your IDE (Eclipse or Android Studio)

There are also LogCat apps on the Play Store, such as aLogCat, that will display the contents of LogCat. However, for security and privacy reasons, on Jelly Bean and higher devices, such apps will only be able to show you *their* LogCat entries, not those from the system, your app, or anyone else. Hence, for development purposes, it is better to use one of the other alternatives outlined above.

LogCat in Android Studio

The LogCat view is available at any time, from pretty much anywhere in Android Studio, by means of clicking on the Android tool window entry, usually docked at the bottom of your IDE window:

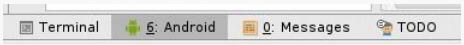


Figure 134: Minimized Tool Windows in Android Studio, Showing Android Tool Window Entry

Tapping on that will bring up some Android-specific logs in an "Android DDMS" tool window, with a tab for "Devices | logcat":

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

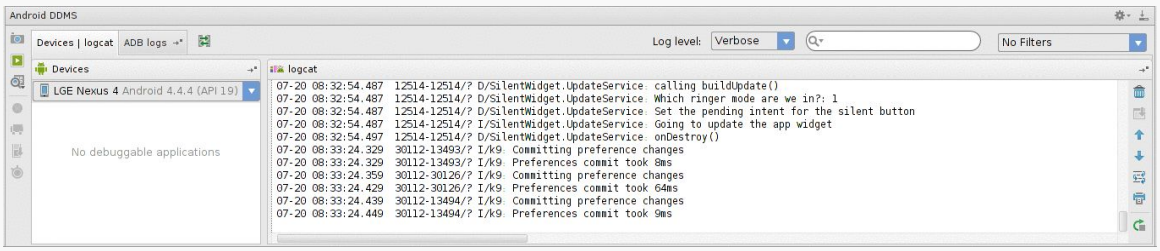


Figure 135: Android DDMS Tool Window, Showing LogCat

LogCat will show your stack traces, diagnostic information from the operating system, and anything you wish to include via calls to static methods on the `android.util.Log` class. For example, `Log.e()` will log a message at error severity, causing it to be displayed in red.

If you want to send something from LogCat to somebody else, such as via an issue tracker, just highlight the text and copy it to the clipboard, as you would with any text editor.

The "trash can" icon atop the tool strip on the right is the "clear log" tool. Clicking it will appear to clear LogCat. It definitely clears your LogCat *view*, so you will only see messages logged after you cleared it. Note, though, that this does not actually clear the logs from the device or emulator.

In addition, you can:

- Use the "Log level" drop-down to filter lines based on severity, where messages for your chosen severity or higher will be displayed
- Use the search field to the right of the "Log level" drop-down to filter items based on a search string
- Set up more permanent filters via the drop-down to the right of the search field

LogCat in Eclipse

The LogCat view is available at any time, from pretty much anywhere in Eclipse, by means of clicking on the LogCat icon in the status bar of your Eclipse window:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

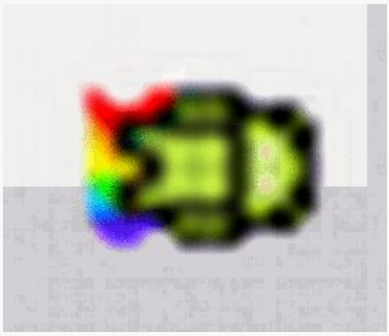


Figure 136: Scaled Up Rendition of LogCat Icon

LogCat will show your stack traces, diagnostic information from the operating system, and anything you wish to include via calls to static methods on the `android.util.Log` class. For example, `Log.e()` will log a message at error severity, causing it to be displayed in red.

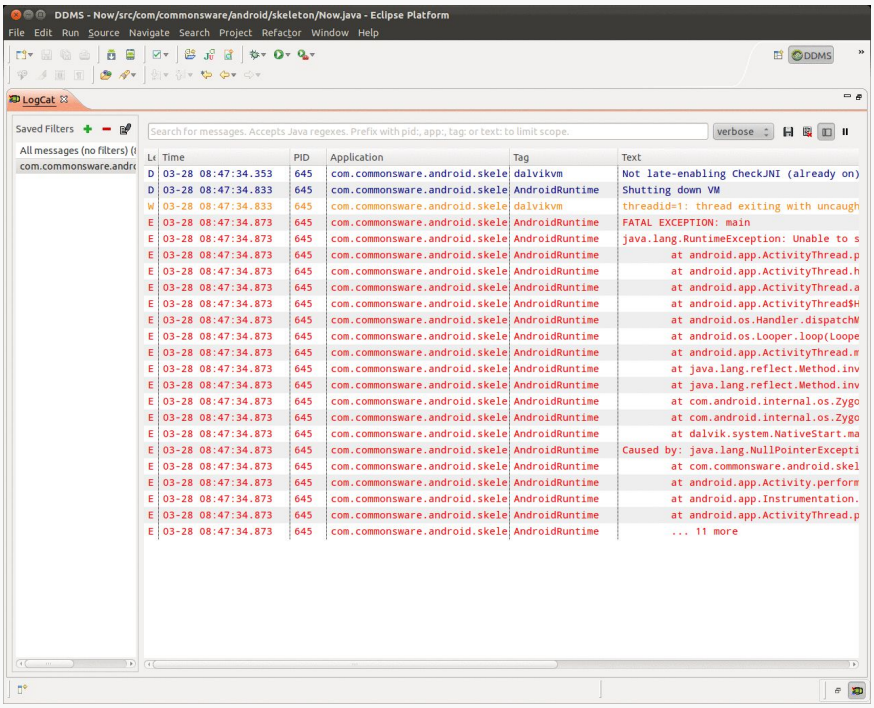


Figure 137: Eclipse Window with LogCat View Maximized

By default, when developing your app, if your app crashes, LogCat will display messages from your app alone, via a filter on the left, with the name of your app's package (e.g., `com.commonsware.android.skeleton`). Switching the filter to "All messages (no filters)" will show all LogCat messages, regardless of origin.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

There is a scrollbar towards the bottom of the main log area that will let you see more of your stack trace:

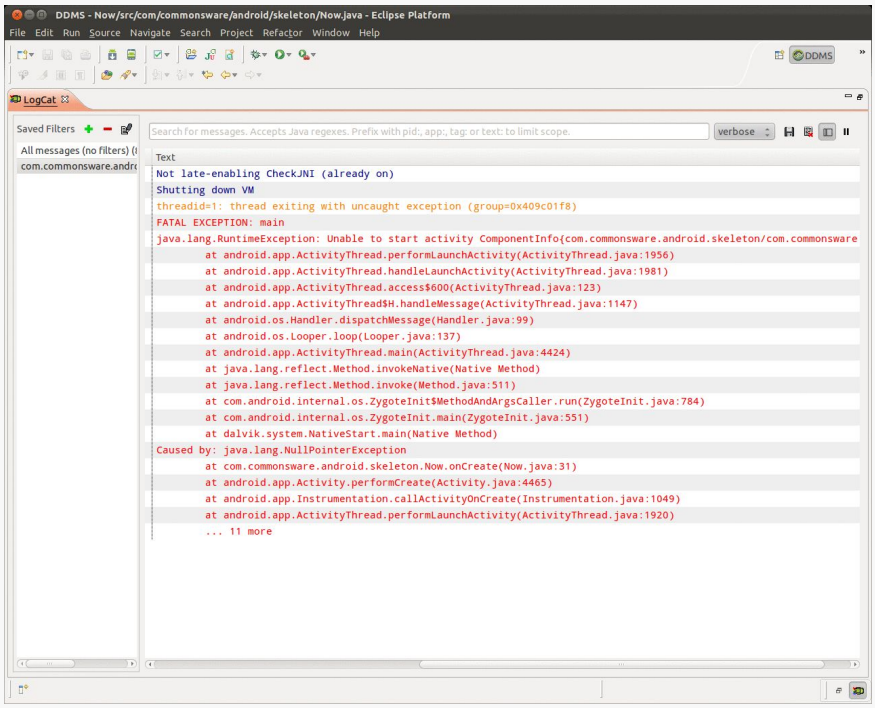


Figure 138: Eclipse Window with LogCat View Scrolled Right

Your stack trace will typically consist of two or more "stanzas". Your own code will typically be in the last of these. So, in the screenshot above, we have `java.lang.RuntimeException: Unable to start activity...`, followed by `Caused by: java.lang.NullPointerException`, as a pair of stanzas. The point where our code crashed shows up in that second stanza (`at com.commonsware.android.skeleton.Now.onCreate(Now.java:31)`).

If you double-click on a line in the stack trace corresponding with your code, you will be taken to a Java editor on that source file and line, so you can see what code triggered the exception.

If you wish to save one of these stack traces as a file, to attach to an issue in an issue tracker or something, highlight the lines you want in LogCat (click on the first line, then `<Shift>`-click on the last line), then click on the "Export Selected Items to Text File" icon (looks like a 3.5-inch floppy disk or a classic "save" icon). This will bring up your platform's "Save As" dialog, where you can specify where to write out the file.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The icon immediately to the right is the "clear" icon:

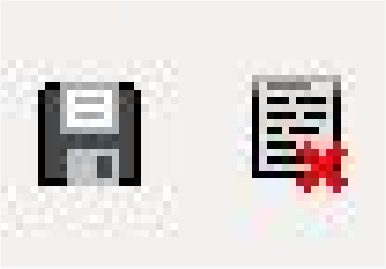


Figure 139: LogCat Save and Clear Icons

Clicking it will appear to clear LogCat. It definitely clears your LogCat *view*, so you will only see messages logged after you cleared it. Note, though, that this does not actually clear the logs from the device or emulator.

The Case of the Confounding Class Cast

If you crash, the stack trace might suggest that there is a problem tied to your resources. One common flavor of this is a `ClassCastException` when you call `findViewById()`. For example, you might call `(Button)findViewById(R.id.button)`, yet get a `ClassCastException: android.widget.LinearLayout` as a result, indicating that while you thought your `findViewById()` call would return a `Button`, it really returned a `LinearLayout`.

Often times, this is not your fault. Sometimes, the `R` values get out of sync with pre-compiled classes from previous builds. This most often occurs just after you change your mix of resources (e.g., add a new layout).

To resolve this, you need to clean your project:

- In Android Studio, choose "Build > Clean Project" from the main menu
- In Eclipse, select the project, then choose Project > Clean from the main menu

So, if you get a strange crash that seems like it might be related to resources, clean your project. If the problem goes away, you are set — if the problem persists, you will need to do a bit more debugging.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Point Break

One of the hallmarks of Java IDEs is the ability to do real-time debugging, using breakpoints and the like. In that respect, Android Studio and Eclipse work for Android apps in the same way that IntelliJ IDEA and Eclipse work for Java apps. You can debug on an emulator or any Android device for which you enabled USB debugging (as you may have done in Tutorial #1).

Lacking any Android Studio-specific documentation, you will wind up referring to the documentation for IntelliJ IDEA to learn how to use its debugger. Similarly, you can turn to the documentation for Eclipse for details of how to use the Eclipse debugger.

Note that if you set up Eclipse to catch all unhandled exceptions, those exceptions will not be logged to LogCat unless you allow execution to proceed past the point of the exception. While this may not matter much to you during development, the LogCat stack trace is often easier for other developers to read, away from your Eclipse environment. So, if you wish to post a stack trace on an issue or on a support forum (e.g., Stack Overflow), use the LogCat stack trace.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

LinearLayout and the Box Model

`LinearLayout` represents Android's approach to a box model — widgets or child containers are lined up in a column or row, one after the next.

Some GUI toolkits use the box as their primary unit of layout. If you want, you can use `LinearLayout` in much the same way, eschewing some of the other containers. Getting the visual representation you want is mostly a matter of identifying where boxes should nest and what properties those boxes should have, such as alignment with respect to other boxes.

Concepts and Properties

To configure a `LinearLayout`, you have four main areas of control besides the container's contents: the orientation, the fill model, the weight, the gravity.

Orientation

Orientation indicates whether the `LinearLayout` represents a row or a column. Just add the `android:orientation` property to your `LinearLayout` element in your XML layout, setting the value to be `horizontal` for a row or `vertical` for a column.

The orientation can be modified at runtime by invoking `setOrientation()` on the `LinearLayout`, supplying it either `HORIZONTAL` or `VERTICAL`.

Fill Model

The point behind a `LinearLayout` — or any of the Android container classes – is to organize multiple widgets. Part of organizing those widgets is determining how much space each gets.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`LinearLayout` takes an "eldest child wins" approach towards allocating space. So, if we have a `LinearLayout` with three children, the first child will get its requested space. The second child will get its requested space, if there is enough room remaining, and likewise for the third child. So if the first child asks for all the space (e.g., this is a `horizontal` `LinearLayout` and the first child has `android:layout_width="match_parent"`), the second and third children will wind up with zero width.

Weight

But, what happens if we have two or more widgets that should split the available free space? For example, suppose we have two multi-line fields in a column, and we want them to take up the remaining space in the column after all other widgets have been allocated their space.

To make this work, in addition to setting `android:layout_width` (for rows) or `android:layout_height` (for columns), you must also set `android:layout_weight`. This property indicates what proportion of the free space should go to that widget. If you set `android:layout_weight` to be the same non-zero value for a pair of widgets (e.g., `1`), the free space will be split evenly between them. If you set it to be `1` for one widget and `2` for another widget, the second widget will use up twice the free space that the first widget does. And so on.

The weight for a widget is zero by default.

Another pattern for using weights is if you want to allocate sizes on a percentage basis. To use this technique for, say, a horizontal layout:

1. Set all the `android:layout_width` values to be `0` for the widgets in the layout
2. Set the `android:layout_weight` values to be the desired percentage size for each widget in the layout
3. Make sure all those weights add up to `100`

If you want to have space left over, not allocated to any widget, you can add an `android:weightSum` attribute to the `LinearLayout`, and ensure that the sum of the `android:layout_weight` attributes of the children are less than that sum. The children will each get space allocated based upon the ratio of their `android:layout_weight` compared to the `android:weightSum`, not compared to the sum of the weights. And there will be empty space that takes up the rest of the room not allocated to the children.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To see android:layout_weight in action, take a look at the Containers/LinearPercent sample project. Here, we have a res/layout/main.xml file containing a vertical LinearLayout with three Button widgets as children:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <Button
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="50"
    android:text="@string/fifty_percent"/>

  <Button
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="30"
    android:text="@string/thirty_percent"/>

  <Button
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="20"
    android:text="@string/twenty_percent"/>

</LinearLayout>
```

Each of the three Button widgets declares its height to be 0dip. However, each also has an android:layout_weight attribute, with the top Button requesting a weight of 50, the middle Button a weight of 30, and the bottom Button a weight of 20.

The result is that the Button widgets' heights are allocated based solely upon those weights:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

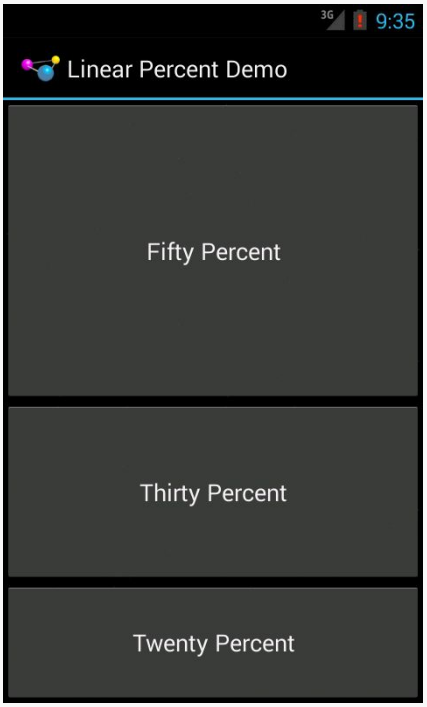


Figure 140: The LinearPercent Sample Application

Gravity

By default, everything in a `LinearLayout` is left- and top-aligned. So, if you create a row of widgets via a horizontal `LinearLayout`, the row will start flush on the left side of the screen.

If that is not what you want, you need to specify a gravity. Unlike the physical world, Android has two types of gravity: the gravity of a widget within a `LinearLayout`, and the gravity of the contents of a widget or container.

The `android:gravity` property of some widgets and containers — which also can be defined via `setGravity()` in Java — tells Android to slide the contents of the widget or container in a particular direction. For example, `android:gravity="right"` says to slide the contents of the widget to the right; `android:gravity="right|bottom"` says to slide the contents of the widget to the right and the bottom.

Here, "contents" varies. `TextView` supports `android:gravity`, and the "contents" is the text held within the `TextView`. `LinearLayout` supports `android:gravity`, and the "contents" are the widgets inside the container. And so on.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Children of a `LinearLayout` also have the option of specifying `android:layout_gravity`. Here, the child is telling the `LinearLayout` "if there is room, please slide me (and me alone) in this direction". However, this only works in the direction *opposite* the orientation of the `LinearLayout` – the children of a `vertical` `LinearLayout` can use `android:layout_gravity` to control their positioning horizontally (`left` or `right`), but not vertically.

For a row of widgets, the default is for them to be aligned so their texts are aligned on the baseline (the invisible line that letters seem to "sit on"), though you may wish to specify a gravity of `center_vertical` to center the widgets along the row's vertical midpoint.

Android Studio Graphical Layout Editor

The `LinearLayout` container can be found in the "Layouts" portion of the Palette of the Android Studio graphical layout editor:

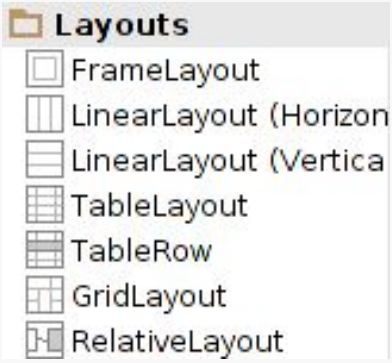


Figure 141: Layouts Palette in Android Studio Graphical Layout Editor

You can drag either the "LinearLayout (Vertical)" or "LinearLayout (Horizontal)" into a layout XML resource, then start dragging in children to go into the container.

When your `LinearLayout` is the selected widget, a few new toolbar buttons will appear over the preview:

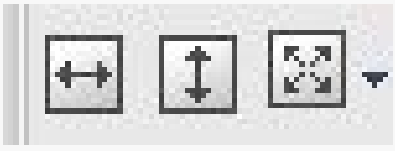


Figure 142: LinearLayout Toolbar Buttons

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The left two buttons toggle the width and height between `match_parent` and `wrap_content`, while the third button changes the gravity of the `LinearLayout`.

When one of the *children* of the `LinearLayout` is the selected widget, the toolbar changes:

Figure 143: LinearLayout Toolbar Buttons, For Selected Child

From left to right, the buttons:

- Toggle the parent `LinearLayout` between horizontal and vertical orientations
- Align the child widgets' baselines (where a "baseline" is the invisible line that text appears to sit upon)
- Change the `android:layout_gravity` value for the child
- Toggle the width between `match_parent` and `wrap_content`
- Toggle the height between `match_parent` and `wrap_content`
- Distribute the weights of the selected children evenly
- Assign this widget all of the weight, at the expense of other children of the `LinearLayout`
- Set the weight to a specific value
- Clear the weights from the children

The Properties pane for the selected widget also allows you to get to the `LinearLayout` container to make adjustments to its attributes.

Eclipse Graphical Layout Editor

The `LinearLayout` container can be found in the "Layouts" portion of the Palette of the Eclipse graphical layout editor:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

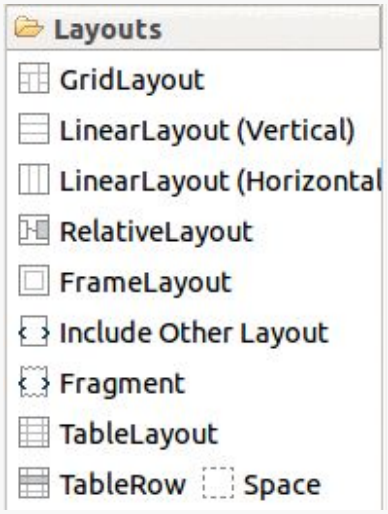


Figure 144: Layouts Palette in Eclipse Graphical Layout Editor

You can drag either the "LinearLayout (Vertical)" or "LinearLayout (Horizontal)" into a layout XML resource, then start dragging in children to go into the container.

When your `LinearLayout` is the selected widget, a toolbar will appear over the preview:

Figure 145: LinearLayout Toolbar in Eclipse Graphical Layout Editor

The left two buttons toggle your `LinearLayout` between vertical and horizontal modes. The two immediately to the right of the divider toggle the width and height between `match_parent` and `wrap_content`.

When one of the *children* of the `LinearLayout` is the selected widget, the toolbar changes:

Figure 146: LinearLayout Contents Toolbar in Eclipse Graphical Layout Editor

The left two buttons still toggle the orientation of the `LinearLayout`. The width and height buttons to their right toggle the width and height of the selected widget.

The right-most six buttons, from left to right, allow you to:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Change the margins on the selected widget
- Change the gravity of the selected widget
- Give all widgets in the `LinearLayout` equal weight
- Give the selected widget all the weight
- Manually assign the weight to the selected widget
- Clear all weights from all widgets in the `LinearLayout`

The Properties pane for the selected widget also allows you to get to the `LinearLayout` container to make adjustments to its attributes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Other Common Widgets and Containers

In the chapter on basic widgets, we left out all of the classic "two-state" widgets, such as checkboxes and radio buttons. We will examine those and other related widgets in this chapter.

Beyond `LinearLayout`, Android supports a range of containers providing different layout rules. In this chapter, we will look at two other commonly-used containers: `RelativeLayout` (a rule-based model) and `TableLayout` (the grid model), along with `ScrollView` and `HorizontalScrollView`, containers that allow their contents to scroll. We will examine all of these containers in this chapter as well.

Just a Box to Check

The classic checkbox has two states: checked and unchecked. Clicking the checkbox toggles between those states to indicate a choice (e.g., "Add rush delivery to my order").

In Android, there is a `CheckBox` widget to meet this need. It has `TextView` as an ancestor, so you can use `TextView` properties like `android:textColor` to format the widget.

Within Java, you can invoke:

1. `isChecked()` to determine if the checkbox has been checked
2. `setChecked()` to force the checkbox into a checked or unchecked state
3. `toggle()` to toggle the checkbox as if the user clicked upon it

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Also, you can register a listener object (in this case, an instance of `OnCheckedChangeListener`) to be notified when the state of the checkbox changes.

For example, from the `Basic/CheckBox` sample project, here is a simple checkbox layout:

```
<?xml version="1.0" encoding="utf-8"?>
<CheckBox xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/check"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/unchecked"/>
```

The corresponding `CheckBoxDemo.java` retrieves and configures the behavior of the checkbox:

```
package com.commonsware.android.checkbox;

import android.app.Activity;
import android.os.Bundle;
import android.widget.CheckBox;
import android.widget.CompoundButton;

public class CheckBoxDemo extends Activity implements
    CompoundButton.OnCheckedChangeListener {
  CheckBox cb;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    cb=(CheckBox)findViewById(R.id.check);
    cb.setOnCheckedChangeListener(this);
  }

  public void onCheckedChanged(CompoundButton buttonView,
                               boolean isChecked) {
    if (isChecked) {
      cb.setText(R.string.checked);
    }
    else {
      cb.setText(R.string.unchecked);
    }
  }
}
```

Note that the activity serves as its own listener for checkbox state changes since it implements the `OnCheckedChangeListener` interface (set via `cb.setOnCheckedChangeListener(this)`). The callback for the listener is

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`onCheckedChanged()`, which receives the checkbox whose state has changed and what the new state is. In this case, we update the text of the checkbox to reflect what the actual box contains.

The result? Clicking the checkbox immediately updates its text, as shown below:

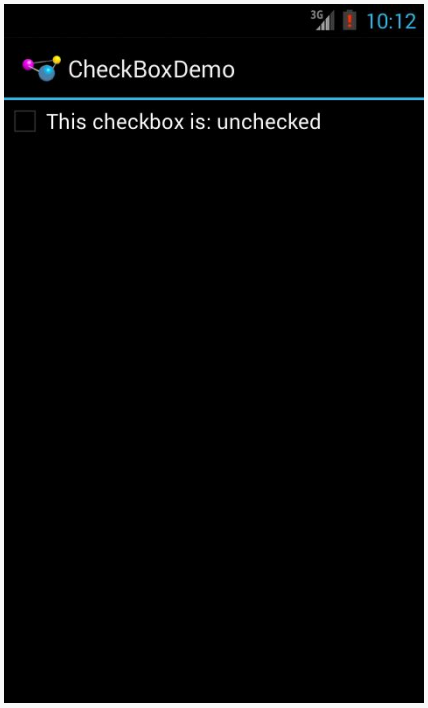


Figure 147: CheckBoxDemo Sample App, in Theme.Holo, with CheckBox Unchecked

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

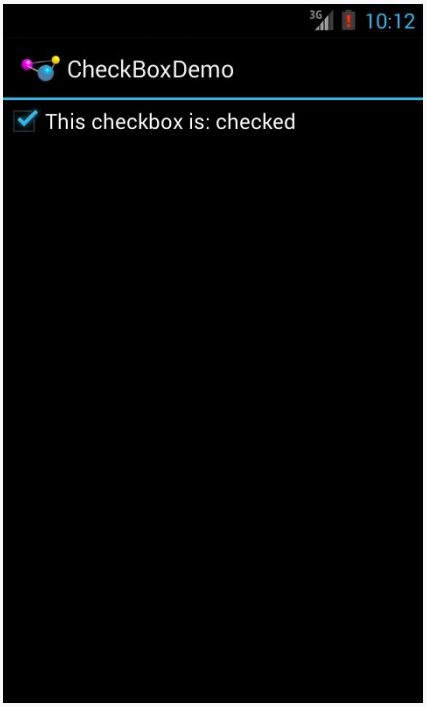


Figure 148: CheckBoxDemo Sample App, in Theme.Holo, with CheckBox Checked

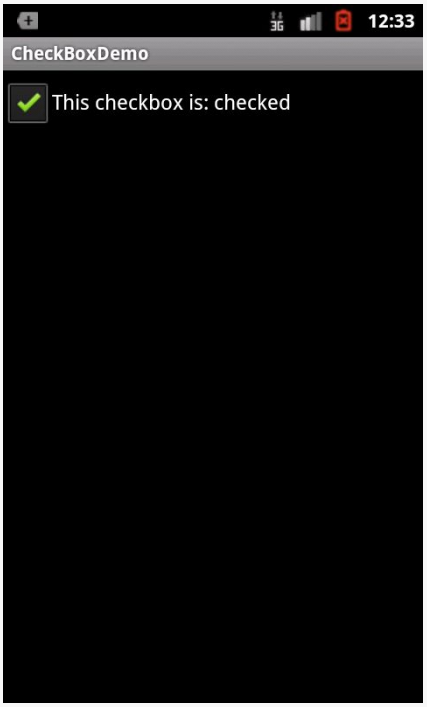


Figure 149: CheckBoxDemo Sample App, in Theme, with CheckBox Checked

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

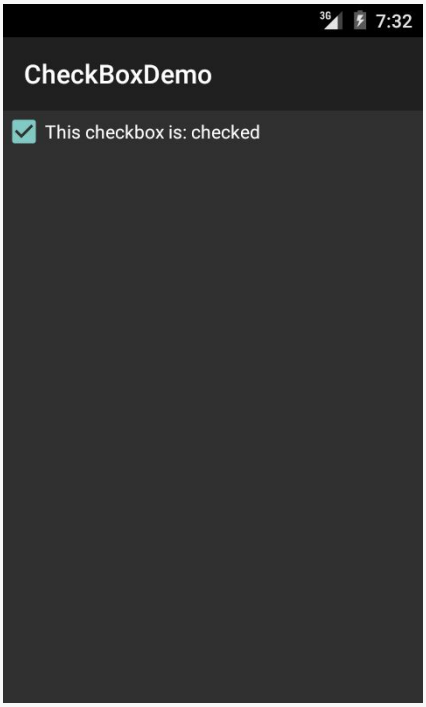


Figure 150: CheckBoxDemo Sample App, in Theme.Material, with CheckBox Checked

Android Studio Graphical Layout Editor

The `CheckBox` widget can be found in the "Widgets" portion of the Palette in the Android Studio Graphical Layout editor:

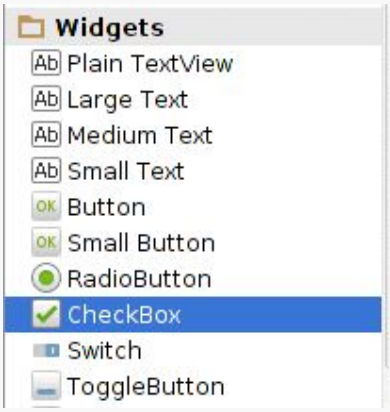


Figure 151: Widgets Palette, CheckBox Shown Highlighted

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You can drag it into the layout and configure it as desired using the Properties pane. As `CheckBox` inherits from `TextView`, most of the settings are the same as those you would find on a regular `TextView`.

Eclipse Graphical Layout Editor

The `CheckBox` widget appears in the "Form Widgets" section of the Palette in the Graphical Layout editor. You can drag it into the layout and configure it as desired using the Properties pane. As `CheckBox` inherits from `TextView`, most of the settings are the same as those you would find on a regular `TextView`.

Don't Like Checkboxes? How About Toggles or Switches?

A similar widget to `CheckBox` is `ToggleButton`. Like `CheckBox`, `ToggleButton` is a two-state widget that is either checked or unchecked. However, `ToggleButton` has a distinct visual appearance:

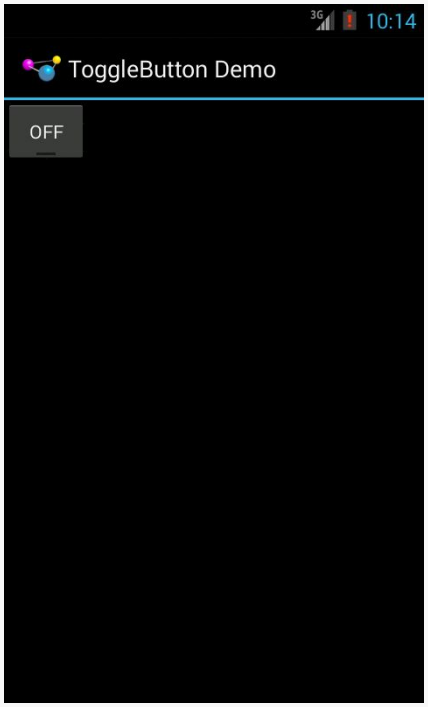


Figure 152: ToggleButtonDemo Sample, Unchecked, in Theme.Holo

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

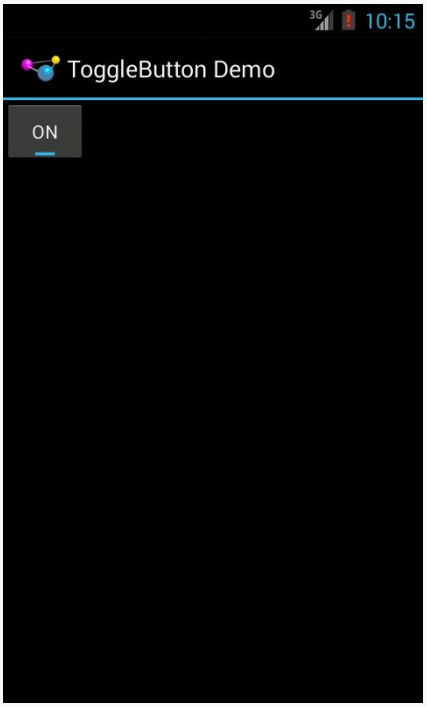


Figure 153: ToggleButtonDemo Sample, Checked, in Theme.Holo

Otherwise, `ToggleButton` behaves much like `CheckBox`. You can put it in a layout file, as seen in the `Basic/ToggleButton` sample:

```
<?xml version="1.0" encoding="utf-8"?>
<ToggleButton xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toggle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

You can also set up an `OnCheckedChangeListener` to be notified when the user changes the state of the `ToggleButton`.

Similarly, Android has a `Switch` widget, showing the state via a small "ON/OFF" slider:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

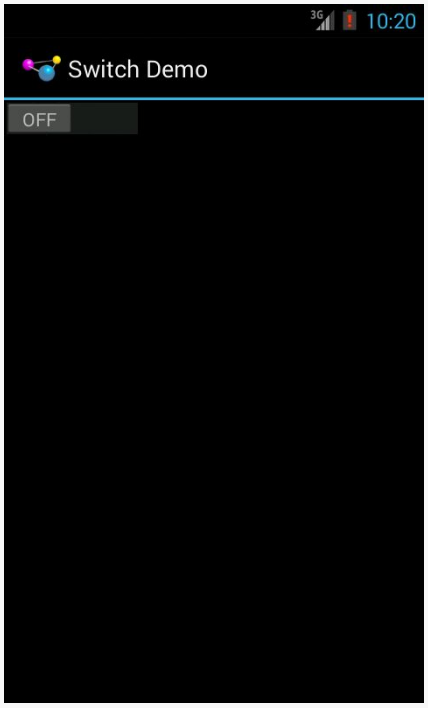


Figure 154: SwitchDemo Sample, Unchecked, in Theme.Holo

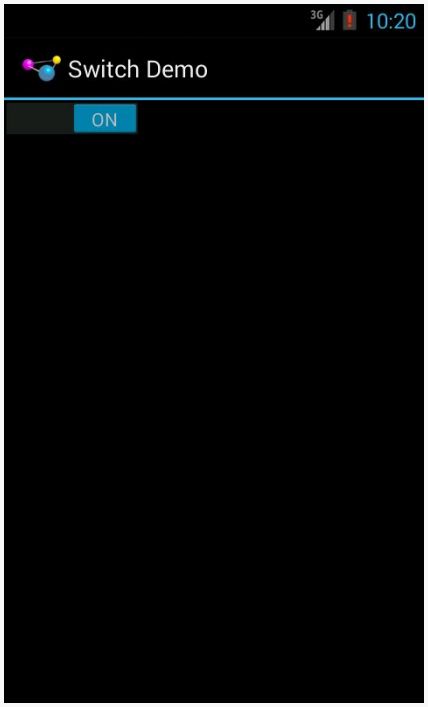


Figure 155: SwitchDemo Sample, Checked, in Theme.Holo

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

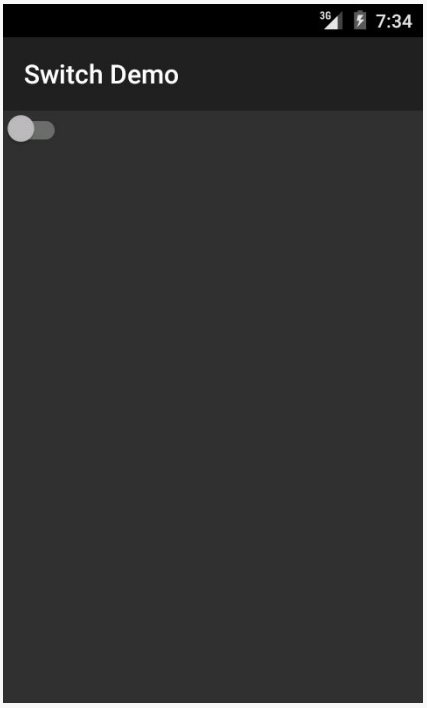


Figure 156: SwitchDemo Sample, Unchecked, in Theme.Material

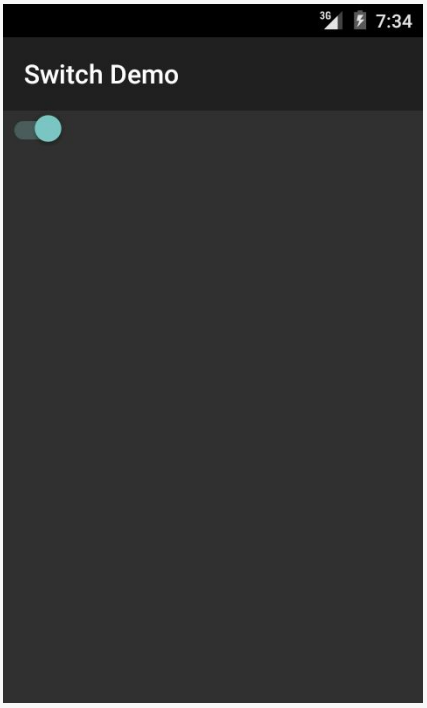


Figure 157: SwitchDemo Sample, Checked, in Theme.Material

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`Switch`, like `CheckBox` and `ToggleButton`, inherits from `CompoundButton`, and therefore shares a common API, for methods like `toggle()`, `isChecked()`, and `setChecked()`. And, as with the others, you can put it in a layout file, as seen in the `Basic/Switch` sample:

```
<?xml version="1.0" encoding="utf-8"?>
<Switch xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toggle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

The biggest limitation with `Switch` is that it was only added to the Android SDK in API Level 14. If your `minSdkVersion` is set to 14 or higher, you are welcome to use `Switch`. If your `minSdkVersion` is set to something lower than 14, though, you will either need to choose something else or get into more complicated scenarios, like using a library that offers a backport of `Switch`. We will cover those more complicated scenarios later in the book; for now, it is simplest to only use `Switch` if your `minSdkVersion` is set to 14 or higher.

Android Studio Graphical Layout Editor

The `ToggleButton` and `Switch` widgets can be found in the "Widgets" portion of the Palette in the Android Studio Graphical Layout editor, just beneath the `CheckBox` widget:

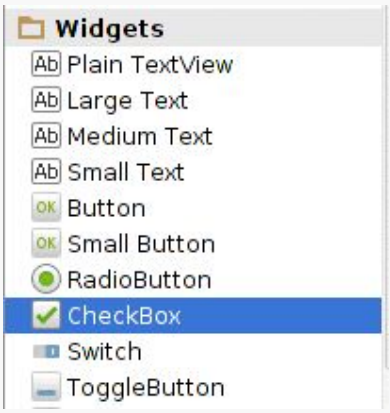


Figure 158: Widgets Palette, ToggleButton and Switch At Bottom

You can drag either widget into the layout and configure it as desired using the Properties pane.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Eclipse Graphical Layout Editor

Like `CheckBox`, the `ToggleButton` and `Switch` widgets appear in the "Form Widgets" section of the Palette in the Graphical Layout editor. You can drag either widget into the layout and configure it as desired using the Properties pane.

Turn the Radio Up

As with other implementations of radio buttons in other toolkits, Android's radio buttons are two-state, like checkboxes, but can be grouped such that only one radio button in the group can be checked at any time.

`CheckBox`, `ToggleButton`, `Switch`, and `RadioButton` all inherit from `CompoundButton`, which in turn inherits from `TextView`. Hence, all the standard `TextView` properties for font face, style, color, etc. are available for controlling the look of radio buttons. Similarly, you can call `isChecked()` on a `RadioButton` to see if it is selected, `toggle()` to change its checked state, and so on, like you can with a `CheckBox`.

Most times, you will want to put your `RadioButton` widgets inside of a `RadioGroup`. The `RadioGroup` is a `LinearLayout` that indicates a set of radio buttons whose state is tied, meaning only one button out of the group can be selected at any time. If you assign an `android:id` to your `RadioGroup` in your XML layout, you can access the group from your Java code and invoke:

1. `check()` to check a specific radio button via its ID (e.g., `group.check(R.id.radio1)`)
2. `clearCheck()` to clear all radio buttons, so none in the group are checked
3. `getCheckedRadioButtonId()` to get the ID of the currently-checked radio button (or `-1` if none are checked)

Note that the mutual-exclusion feature of `RadioGroup` only applies to `RadioButton` widgets that are immediate children of the `RadioGroup`. You cannot have other containers between the `RadioGroup` and its `RadioButton` widgets.

For example, from the `Basic/RadioButton` sample application, here is an XML layout showing a `RadioGroup` wrapping a set of `RadioButton` widgets:

```
<?xml version="1.0" encoding="utf-8"?>
<RadioGroup
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
    <RadioButton android:id="@+id/radio1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/rock" />

    <RadioButton android:id="@+id/radio2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/scissors" />

    <RadioButton android:id="@+id/radio3"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/paper" />
</RadioGroup>
```

Using the stock Android-generated Java for the project and this layout, you get:

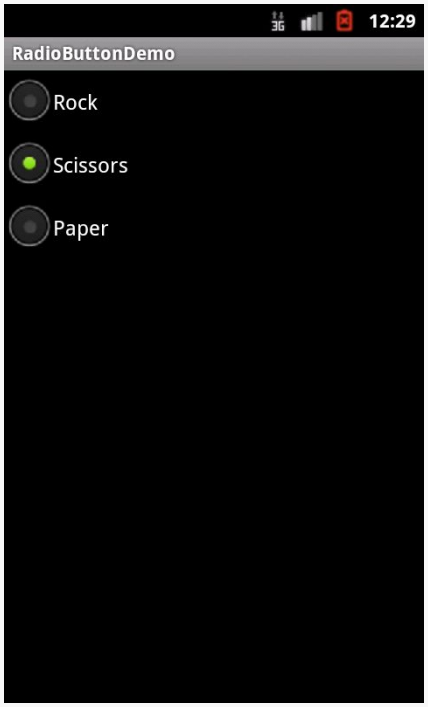


Figure 159: RadioButtonDemo, with "Scissors" Checked, in Theme

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

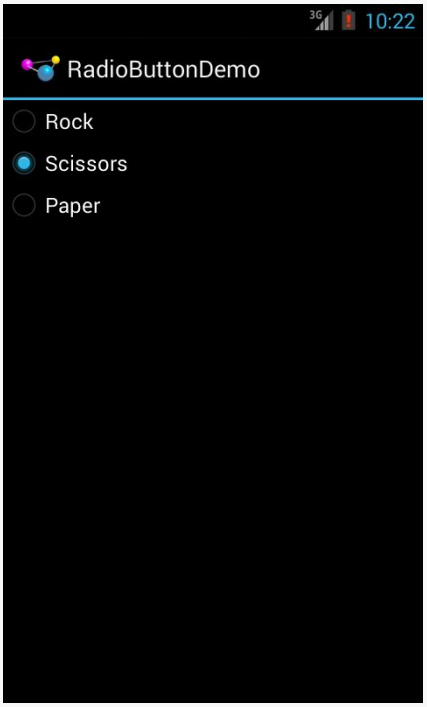


Figure 160: RadioButtonDemo, with "Scissors" Checked, in Theme.Holo

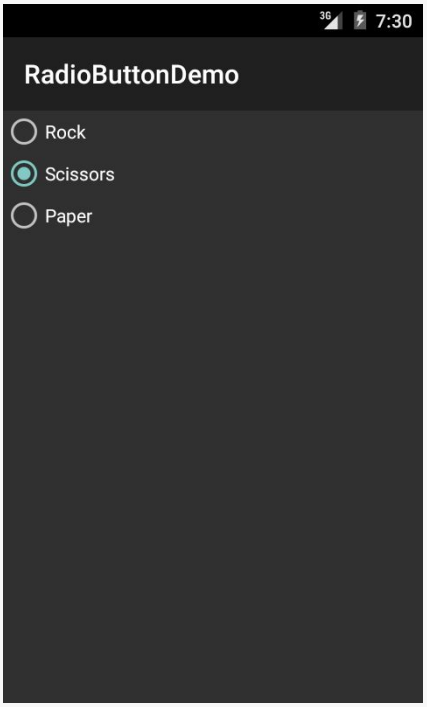


Figure 161: RadioButtonDemo, with "Scissors" Checked, in Theme.Material

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that the radio button group is initially set to be completely unchecked at the outset. To preset one of the radio buttons to be checked, use either `setChecked()` on the `RadioButton` or `check()` on the `RadioGroup` from within your `onCreate()` callback in your activity. Alternatively, you can use the `android:checked` attribute on one of the `RadioButton` widgets in the layout file.

Android Studio Graphical Layout Editor

The `RadioGroup` container can be found in the "Containers" portion of the Palette in the Android Studio Graphical Layout editor:

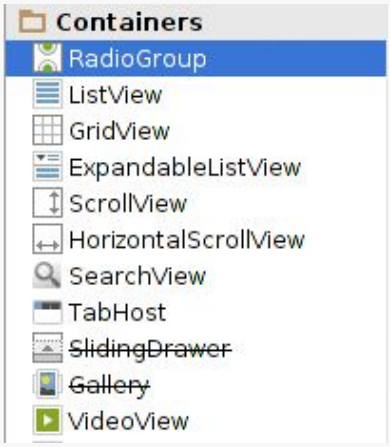


Figure 162: Widgets Palette, RadioGroup Highlighted

Dragging a `RadioGroup` into the preview works much like dragging a `LinearLayout` into the preview. You get a box into which you can drag other widgets, such as the `RadioButton` found in the "Widgets" section of the Palette.

Eclipse Graphical Layout Editor

Both `RadioButton` and `RadioGroup` appear in the "Form Widgets" section of the Palette in the Graphical Layout editor. The `RadioButton` widget has a radio button with the text "RadioButton" to the right. The `RadioGroup` widget looks like three radio buttons (sans text) side-by-side.

Since `RadioGroup` extends `LinearLayout`, when you drag it into the layout, you will get the same sorts of options as a vertical `LinearLayout`, such as setting the gravity. Note, though, that dragging a `RadioGroup` into a layout automatically gives you three

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

RadioButton child widgets — a departure from any other container in the Palette. You can configure those RadioButton widgets, delete them, add more, etc.

All Things Are Relative

RelativeLayout, as the name suggests, lays out widgets based upon their relationship to other widgets in the container and the parent container. You can place Widget X below and to the left of Widget Y, or have Widget Z's bottom edge align with the bottom of the container, and so on.

Concepts and Properties

To make all this work, we need ways to reference other widgets within an XML layout file, plus ways to indicate the relative positions of those widgets.

Positions Relative to Container

The easiest relations to set up are tying a widget's position to that of its container:

1. android:layout_alignParentTop says the widget's top should align with the top of the container
2. android:layout_alignParentBottom says the widget's bottom should align with the bottom of the container
3. android:layout_alignParentLeft says the widget's left side should align with the left side of the container
4. android:layout_alignParentRight says the widget's right side should align with the right side of the container
5. android:layout_centerHorizontal says the widget should be positioned horizontally at the center of the container
6. android:layout_centerVertical says the widget should be positioned vertically at the center of the container
7. android:layout_centerInParent says the widget should be positioned both horizontally and vertically at the center of the container

All of these properties take a simple boolean value (true or false).

Note that the padding of the widget is taken into account when performing these various alignments. The alignments are based on the widget's overall cell (combination of its natural space plus the padding).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Relative Notation in Properties

The remaining properties of relevance to `RelativeLayout` take as a value the identity of a widget in the container. To do this:

- Put identifiers (`android:id` attributes) on all elements that you will need to address
- Reference other widgets using the same identifier value

The first occurrence of an id value should have the plus sign (`@+id/widget_a`); the second and subsequent times that id value is used in the layout file should drop the plus sign (`@id/widget_a`). This allows the build tools to better help you catch typos in your widget `id` values — if you do not have a plus sign for a widget `id` value that has not been seen before, that will be caught at compile time.

For example, if Widget A appears in the `RelativeLayout` before Widget B, and Widget A is identified as `@+id/widget_a`, Widget B can refer to Widget A in one of its own properties via the identifier `@id/widget_a`.

Positions Relative to Other Widgets

There are four properties that control position of a widget *vis-à-vis* other widgets:

1. `android:layout_above` indicates that the widget should be placed above the widget referenced in the property
2. `android:layout_below` indicates that the widget should be placed below the widget referenced in the property
3. `android:layout_toLeftOf` indicates that the widget should be placed to the left of the widget referenced in the property
4. `android:layout_toRightOf` indicates that the widget should be placed to the right of the widget referenced in the property

Beyond those four, there are five additional properties that can control one widget's alignment relative to another:

1. `android:layout_alignTop` indicates that the widget's top should be aligned with the top of the widget referenced in the property
2. `android:layout_alignBottom` indicates that the widget's bottom should be aligned with the bottom of the widget referenced in the property
3. `android:layout_alignLeft` indicates that the widget's left should be aligned with the left of the widget referenced in the property

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

4. `android:layout_alignRight` indicates that the widget's right should be aligned with the right of the widget referenced in the property
5. `android:layout_alignBaseline` indicates that the baselines of the two widgets should be aligned (where the "baseline" is that invisible line that text appears to sit on)

The last one is useful for aligning labels and fields so that the text appears "natural". Since fields have a box around them and labels do not, `android:layout_alignTop` would align the top of the field's box with the top of the label, which will cause the text of the label to be higher on-screen than the text entered into the field.

So, if we want Widget B to be positioned to the right of Widget A, in the XML element for Widget B, we need to include `android:layout_toRightOf = "@id/widget_a"` (assuming `@id/widget_a` is the identity of Widget A).

Example

With all that in mind, let's examine a typical "form" with a field, a label, plus a pair of buttons labeled "OK" and "Cancel".

Here is the XML layout, pulled from the `Containers/Relative` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

  <TextView
    android:id="@+id/label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignBaseline="@+id/entry"
    android:layout_alignParentLeft="true"
    android:layout_marginLeft="4dip"
    android:text="@string/url"/>

  <EditText
    android:id="@id/entry"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentTop="true"
    android:layout_toRightOf="@id/label"
    android:inputType="text"/>

  <Button
    android:id="@+id/ok"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignRight="@id/entry"
    android:layout_below="@id/entry"
    android:text="@string/ok"/>

  <Button
    android:id="@+id/cancel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/ok"
    android:layout_toLeftOf="@id/ok"
    android:text="@string/cancel"/>

</RelativeLayout>
```

First, we open up the `RelativeLayout`. In this case, we want to use the full width of the screen (`android:layout_width = "match_parent"`) and only as much height as we need (`android:layout_height = "wrap_content"`).

Next, we define the label as a `TextView`. We indicate that we want its left edge aligned with the left edge of the `RelativeLayout` (`android:layout_alignParentLeft="true"`) and that we want its baseline aligned with the baseline of the yet-to-be-defined `EditText`. Since the `EditText` has not been declared yet, we use the `+` sign in the ID (`android:layout_alignBaseline="@+id/entry"`).

After that, we add in the field as an `EditText`. We want the field to be to the right of the label, have the field be aligned with the top of the `RelativeLayout`, and for the field to take up the rest of this "row" in the layout. Those are handled by three properties:

1. `android:layout_toRightOf = "@id/label"`
2. `android:layout_alignParentTop = "true"`
3. `android:layout_width = "match_parent"`

Then, the OK button is set to be below the field (`android:layout_below = "@id/entry"`) and have its right side align with the right side of the field (`android:layout_alignRight = "@id/entry"`). The Cancel button is set to be to the left of the OK button (`android:layout_toLeft = "@id/ok"`) and have its top aligned with the OK button (`android:layout_alignTop = "@id/ok"`).

With no changes to the auto-generated Java code, the emulator gives us:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

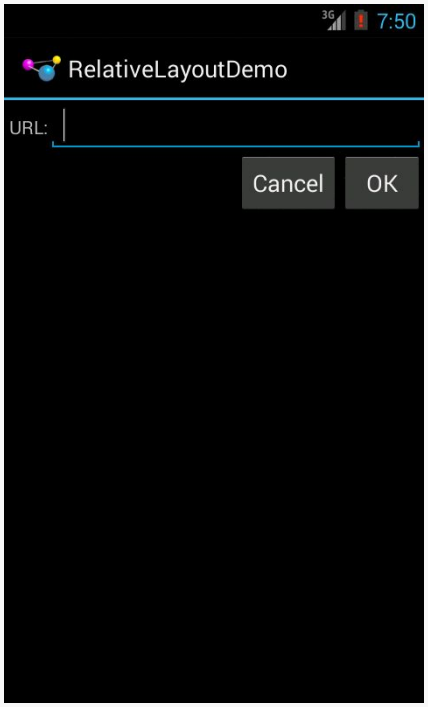


Figure 163: The RelativeLayoutDemo sample application

Overlap

`RelativeLayout` also has a feature that `LinearLayout` lacks — the ability to have widgets overlap one another. Later children of a `RelativeLayout` are "higher in the Z axis" than are earlier children, meaning that later children will overlap earlier children if they are set up to occupy the same space in the layout.

This will be clearer with an example. Here is a layout, from the `Containers/RelativeOverlap` sample, with a `RelativeLayout` holding two `Button` widgets:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:text="@string/big"
    android:textSize="120dip"
    android:textStyle="bold"/>

  <Button
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:text="@string/small"/>

</RelativeLayout>
```

The first `Button` is set to fill the screen. The second `Button` is set to be centered inside the parent, but only take up as much space as is needed for its caption. Hence, the second `Button` will appear to "float" over the first `Button`:

Figure 164: The RelativeOverlap sample application

Both `Button` widgets can still be clicked, though clicking on the smaller `Button` does not also click the bigger `Button`. Your clicks will be handled by the widget on top in the case of an overlap like this.

Android Studio Graphical Layout Editor

You will find `RelativeLayout` in the "Layouts" section of the Palette in the Android Studio Graphical Layout editor. You can drag that into your layout XML resource.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

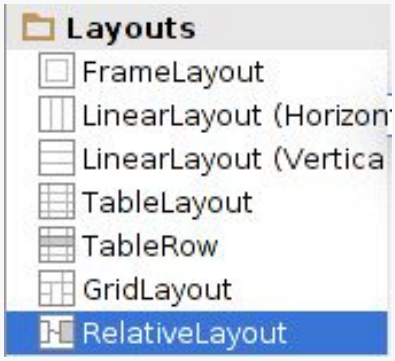


Figure 165: Layouts Section of Palette, RelativeLayout Highlighted

As you drag other widgets into your `RelativeLayout`, you will see a popup indicator of the `RelativeLayout` rules that will be applied if you were to drop the widget at the current mouse pointer position:

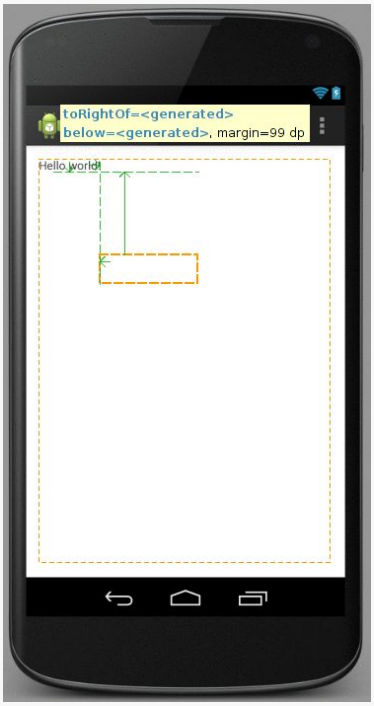


Figure 166: Dragging a Widget in a RelativeLayout

Getting the rules that you *want* may or may not be possible purely through drag-and-drop. You may need to just drop the widget into the `RelativeLayout` and manually adjust the rules, whether by using the Properties pane or by editing the XML directly.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

NOTE: There is a bug in the layout editor that renders the `RelativeLayout` incorrectly on Android Studio, if your preview is set to API Level 21. If you drop the preview to API Level 19, the `RelativeLayout` renders correctly. And, running the app on a device shows that the `RelativeLayout` renders correctly, even on Android 5.0.

Eclipse Graphical Layout Editor

You will find `RelativeLayout` in the "Layouts" section of the Palette in the Eclipse Graphical Layout editor. You can drag that into your layout XML resource.

As with Android Studio, when you drag a widget into a `RelativeLayout`, a popup will indicate the `RelativeLayout` rules that will be applied if you drop the widget at the current location. Sometimes, you cannot get the rules that you really want, forcing you to modify the `RelativeLayout` XML directly via the other editor sub-tab or via the Properties pane to get it set up properly.

Tabula Rasa

If you like HTML tables, you will like Android's `TableLayout`. It allows you to position your widgets in a grid to your specifications. You control the number of rows and columns, which columns might shrink or stretch to accommodate their contents, and so on.

`TableLayout` works in conjunction with `TableRow`. `TableLayout` controls the overall behavior of the container, with the widgets themselves poured into one or more `TableRow` containers, one per row in the grid.

Concepts and Properties

For all this to work, we need to figure out how widgets work with rows and columns, plus how to handle widgets that live outside of rows.

Putting Cells in Rows

Rows are declared by you, the developer, by putting widgets as children of a `TableRow` inside the overall `TableLayout`. You, therefore, control directly how many rows appear in the table.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The number of columns are determined by Android; you control the number of columns in an indirect fashion.

First, there will be at least one column per widget in your longest row. So if you have three rows, one with two widgets, one with three widgets, and one with four widgets, there will be at least four columns.

However, a widget can take up more than one column by including the `android:layout_span` property, indicating the number of columns the widget spans. This is akin to the `colspan` attribute one finds in table cells in HTML:

```
<TableRow>
  <TextView android:text="URL:" />
  <EditText
    android:id="@+id/entry"
    android:layout_span="3"/>
</TableRow>
```

In the above XML layout fragment, the field spans three columns.

Ordinarily, widgets are put into the first available column. In the above fragment, the label would go in the first column (column `0`, as columns are counted starting from `0`), and the field would go into a spanned set of three columns (columns `1` through `3`). However, you can put a widget into a different column via the `android:layout_column` property, specifying the `0`-based column the widget belongs to:

```
<TableRow>
  <Button
    android:id="@+id/cancel"
    android:layout_column="2"
    android:text="Cancel" />
  <Button android:id="@+id/ok" android:text="OK" />
</TableRow>
```

In the preceding XML layout fragment, the Cancel button goes in the third column (column `2`). The OK button then goes into the next available column, which is the fourth column.

Non-Row Children of TableLayout

Normally, `TableLayout` contains only `TableRow` elements as immediate children. However, it is possible to put other widgets in between rows. For those widgets, `TableLayout` behaves a bit like `LinearLayout` with vertical orientation. The widgets

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

automatically have their width set to `match_parent`, so they will fill the same space that the longest row does.

Stretch, Shrink, and Collapse

By default, each column will be sized according to the "natural" size of the widest widget in that column (taking spanned columns into account). Sometimes, though, that does not work out very well, and you need more control over column behavior.

You can place an `android:stretchColumns` property on the `TableLayout`. The value should be a single column number (again, `0`-based) or a comma-delimited list of column numbers. Those columns will be stretched to take up any available space yet on the row. This helps if your content is narrower than the available space.

Conversely, you can place an `android:shrinkColumns` property on the `TableLayout`. Again, this should be a single column number or a comma-delimited list of column numbers. The columns listed in this property will try to word-wrap their contents to reduce the effective width of the column — by default, widgets are not word-wrapped. This helps if you have columns with potentially wordy content that might cause some columns to be pushed off the right side of the screen.

You can also leverage an `android:collapseColumns` property on the `TableLayout`, again with a column number or comma-delimited list of column numbers. These columns will start out "collapsed", meaning they will be part of the table information but will be invisible. Programmatically, you can collapse and un-collapse columns by calling `setColumnCollapsed()` on the `TableLayout`. You might use this to allow users to control which columns are of importance to them and should be shown versus which ones are less important and can be hidden.

You can also control stretching and shrinking at runtime via `setColumnStretchable()` and `setColumnShrinkable()`.

Example

The XML layout fragments shown above, when combined, give us a `TableLayout` rendition of the "form" we created for `RelativeLayout`, with the addition of a divider line between the label/field and the two buttons (found in the `Containers/Table` demo):

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:layout_width="match_parent"
  android:layout_height="match_parent"
    android:stretchColumns="1">

  <TableRow>

    <TextView
      android:layout_marginLeft="4dip"
      android:text="@string/url"/>

    <EditText
      android:id="@+id/entry"
      android:layout_span="3"
      android:inputType="text"/>
  </TableRow>

  <View
    android:layout_height="2dip"
    android:background="#0000FF"/>

  <TableRow>

    <Button
      android:id="@+id/cancel"
      android:layout_column="2"
      android:text="@string/cancel"/>

    <Button
      android:id="@+id/ok"
      android:text="@string/ok"/>
  </TableRow>

</TableLayout>
```

When compiled against the generated Java code and run on the emulator, we get:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

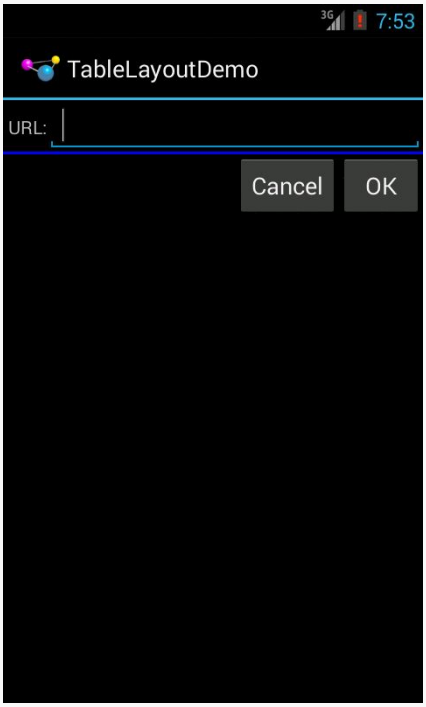


Figure 167: The TableLayoutDemo sample application

Android Studio Graphical Layout Editor

You will find `TableLayout` in the "Layouts" section of the Palette in the Android Studio Graphical Layout editor. You can drag that into your layout XML resource.

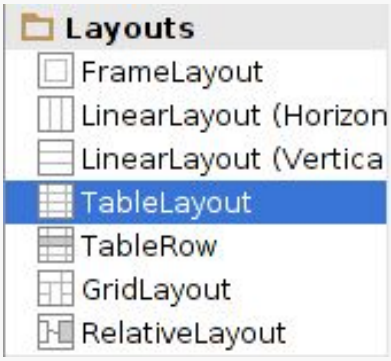


Figure 168: Layouts Section of Palette, TableLayout Highlighted

You might expect that you would then drag in `TableRow` containers as needed for your rows, then drag widgets into the `TableRow` containers. That's not how Android Studio chose to implement the `TableLayout` drag-and-drop support.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Instead, you drag widgets directly into the `TableLayout`, which will switch to a "grid mode", allowing you to indicate where in a table you want this widget to reside as a cell:

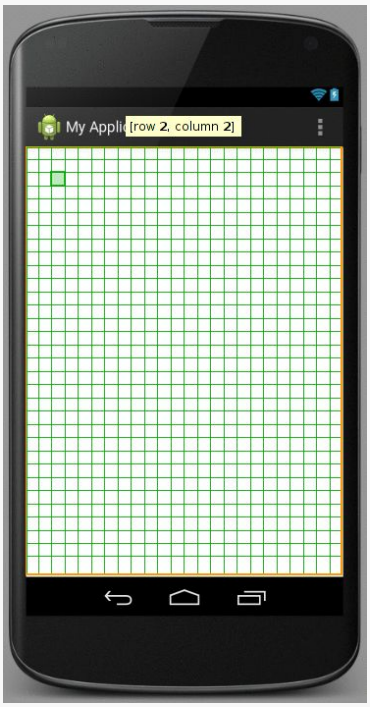


Figure 169: TableLayout Drag-and-Drop in Android Studio

Dropping the widget in a given cell position will:

- Add any necessary `TableRow` widgets, both for the row you requested and any prior rows that were not already set up, and
- Add the widget into the `TableRow`, with an `android:layout_column` value if needed to put the widget into the desired column in that row

For example, if you start with an empty `TableLayout`, then drag a `CheckBox` into "row 2, column 2", you wind up with the following layout XML:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    tools:context=".MyActivity">

    <TableRow
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"></TableRow>

    <TableRow
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"></TableRow>

    <TableRow
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">

        <CheckBox
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="New CheckBox"
            android:id="@+id/checkBox2"
            android:layout_column="2" />
    </TableRow>
</TableLayout>
```

Three `TableRow` widgets were added (for rows 0, 1, and 2), and the `CheckBox` is given `android:layout_column="2"` to put it in column 2.

Eclipse Graphical Layout Editor

You will find `TableLayout` in the "Layouts" section of the Palette in the Eclipse Graphical Layout editor. You can drag that into your layout XML resource and start configuring it via the context menu, notably editing the `android:stretchColumns` and `android:shrinkColumns` values.

In addition, the toolbar above the layout will now sport an add-row button:

Figure 170: Eclipse Layout Toolbar for TableLayout

Clicking that adds a `TableRow` child to the `TableLayout`, though you will not necessarily see a visible change. However, now if you start dragging in other widgets, they will go in that row.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Once you have started to populate the row and can select it, you will get some more toolbar buttons:

Figure 171: Eclipse Layout Toolbar for TableLayout, with Row Selected

The icon immediately to the right of the add-row button will remove the selected row from the table. On the far right side of the toolbar are buttons to allow you to toggle the height and width of the row, plus toggle on and off baseline alignment for the contents of the row (enabled by default).

Scrollwork

Phone screens tend to be small, which requires developers to use some tricks to present a lot of information in the limited available space. One trick for doing this is to use scrolling, so only part of the information is visible at one time, the rest available via scrolling up or down.

`ScrollView` is a container that provides scrolling for its contents. You can take a layout that might be too big for some screens, wrap it in a `ScrollView`, and still use your existing layout logic. It just so happens that the user can only see part of your layout at one time, the rest available via scrolling.

For example, here is a `ScrollView` used in an XML layout file (from the `Containers/Scroll` demo):

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">
  <TableLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="0">
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#000000"/>
      <TextView android:text="#000000"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      <View
        android:layout_height="80dip"
        android:background="#440000" />
      <TextView android:text="#440000"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#884400" />
      <TextView android:text="#884400"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#aa8844" />
      <TextView android:text="#aa8844"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#ffaa88" />
      <TextView android:text="#ffaa88"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#ffffaa" />
      <TextView android:text="#ffffaa"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
    <TableRow>
      <View
        android:layout_height="80dip"
        android:background="#ffffff" />
      <TextView android:text="#ffffff"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
  </TableLayout>
</ScrollView>
```

Without the `ScrollView`, the table would take up at least 560 density-independent pixels (7 rows at 80 dips each, based on the `View` declarations). There may be some devices with screens capable of showing that much information, but many will be

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

smaller. The `ScrollView` lets us keep the table as-is, but only present part of it at a time.

On the stock Android emulator, when the activity is first viewed, you see:

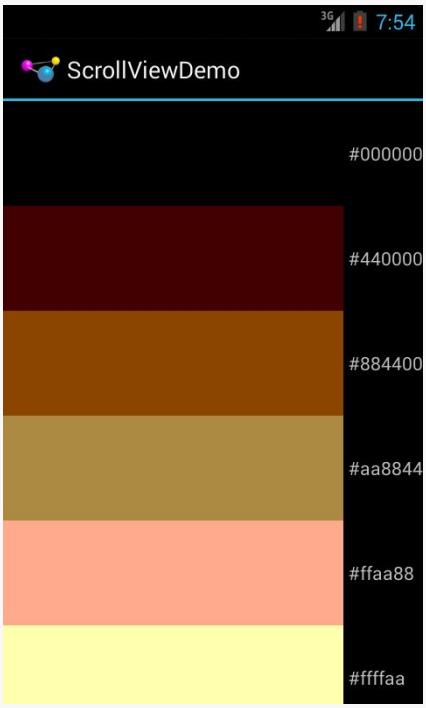


Figure 172: The ScrollViewDemo sample application

Notice how only five rows and part of the sixth are visible. You can scroll up and down to see the remaining rows. Also note how the right side of the content gets clipped by the scrollbar — be sure to put some padding on that side or otherwise ensure your own content does not get clipped in that fashion.

Android also has `HorizontalScrollView`, which works like `ScrollView`... just horizontally. This would be good for forms that might be too wide rather than too tall. Note that `ScrollView` only scrolls vertically and `HorizontalScrollView` only scrolls horizontally.

Also, note that you cannot put scrollable items into a `ScrollView`. For example, a `ListView` widget — which we will see in an upcoming chapter — already knows how to scroll. You do not need to put a `ListView` in a `ScrollView`, and if you were to try, it would not work very well.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

And, a `ScrollView` or `HorizontalScrollView` can only have one child — if you want more than one, wrap the children in a suitable container class (e.g., a `LinearLayout`) and put that inside the `ScrollView` or `HorizontalScrollView`.

Android Studio Graphical Layout Editor

The `ScrollView` and `HorizontalScrollView` widgets appear in the "Containers" section of the Palette in the Graphical Layout editor. You can drag one of these into your layout XML resource, then drag *one* child into it.

Eclipse Graphical Layout Editor

The `ScrollView` and `HorizontalScrollView` widgets appear in the "Composite" section of the Palette in the Graphical Layout editor. You can drag one of these into your layout XML resource, then drag *one* child into it.

Making Progress with ProgressBars

If you are going to fork background threads to do work on behalf of the user, you will want to think about keeping the user informed that work is going on. This is particularly true if the user is effectively waiting for that background work to complete.

The typical approach to keeping users informed of progress is some form of progress bar, like you see when you copy a bunch of files from place to place in many desktop operating systems. Android supports this through the `ProgressBar` widget.

A `ProgressBar` keeps track of progress, defined as an integer, with `0` indicating no progress has been made. You can define the maximum end of the range — what value indicates progress is complete — via `setMax()`. By default, a `ProgressBar` starts with a progress of `0`, though you can start from some other position via `setProgress()`.

If you prefer your progress bar to be indeterminate — meaning that it will show a general animated effect, rather than a specific amount of progress – use `setIndeterminate()`, setting it to `true`.

In your Java code, you can either positively set the amount of progress that has been made (via `setProgress()`) or increment the progress from its current amount (via

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`incrementProgressBy()`). You can find out how much progress has been made via `getProgress()`.

We will see a `ProgressBar` in action in the next chapter, another one of our tutorials.

Visit the Trails!

The trails portion of the book contains a widget catalog, providing capsule descriptions and samples for a number of widgets not described elsewhere in this book.

You might also be interested in `GridLayout`, which is an alternative to the classic `LinearLayout`, `RelativeLayout`, and `TableLayout` containers.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #5 - Making Progress

When we actually get around to opening the digital book for display, there will be a slight delay as the HTML and other assets are read into memory. To help assure the user that their device has not frozen, we will add a `ProgressBar` to our user interface in this tutorial.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Removing The "Hello, World"

Right now, our user interface consists of a highly-sophisticated "Hello, World" string, shown in a `TextView`. While no doubt it is eligible for many design awards, this is not the user interface we need. So, we need to get rid of it.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Double-click on the `res/layout/main.xml` file. This should bring up the graphical layout editor on your initial layout:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

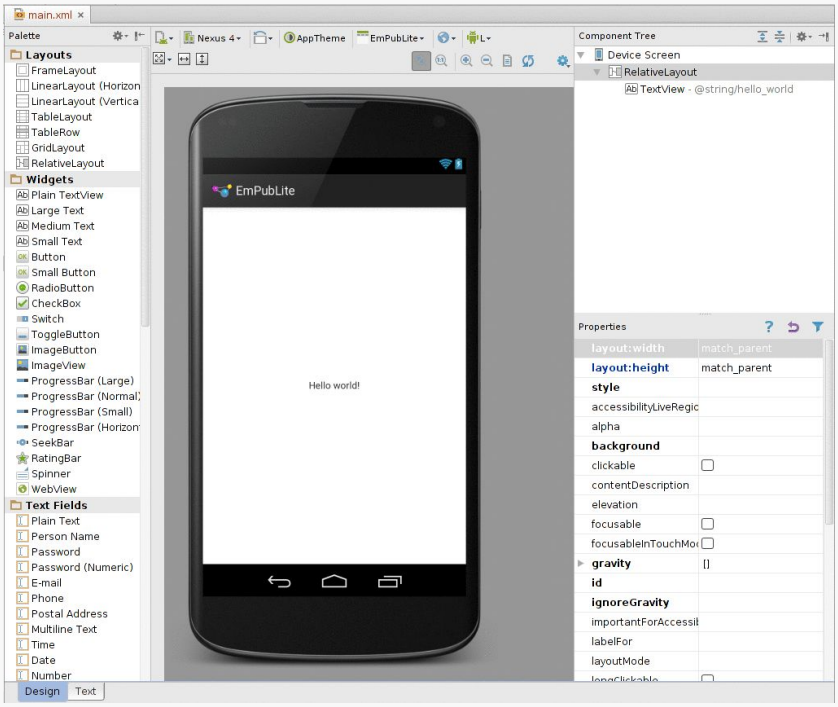


Figure 173: `main.xml` Layout in Android Studio

Click on the "Hello, world!" `TextView` in the middle of that layout to highlight it, then press the Delete key to delete it. At this point, you can save your file.

Also, we no longer need the `hello_world` string resource. To remove it, open the `res/values/strings.xml` file. Find the `<string>` element that has a `name` of `hello_world`, delete that element, and save the file.

The resulting XML should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">EmPub Lite</string>

</resources>
```

Eclipse

Double-click on the `res/layout/main.xml` file in your project in Eclipse's Package Explorer. If you do not have such a file, but you have some other layout resource (e.g., `res/layout/activity_main.xml`), rename it to `main.xml` by right-clicking over

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the file in the Package Explorer and choosing Refactor > Rename from the right-mouse menu. Then double-click on the newly-renamed file.

This will bring up our current user interface:

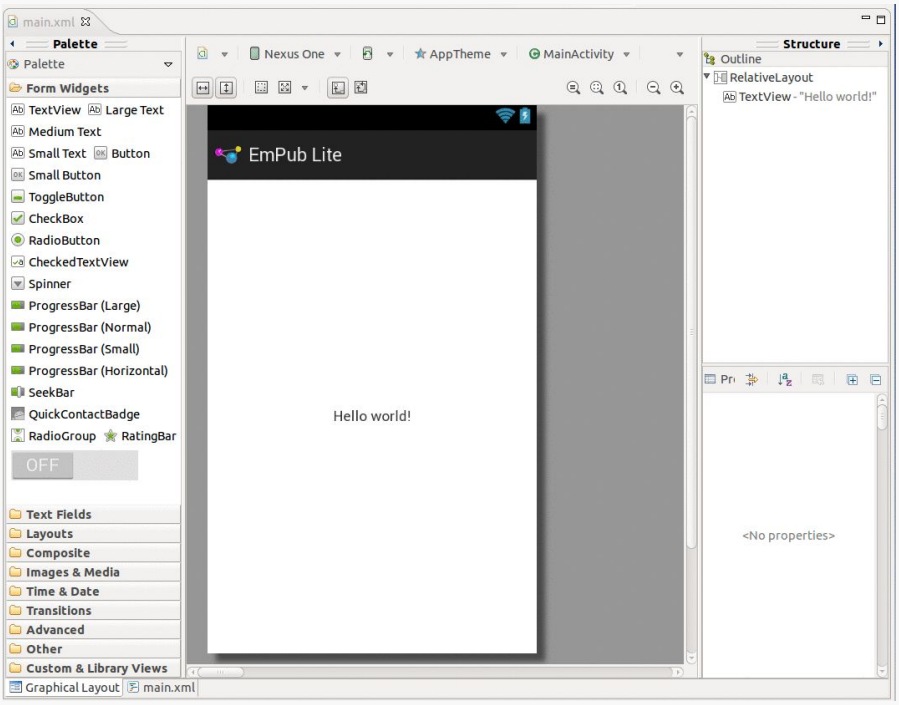


Figure 174: EmPubLiteActivity, in Eclipse

Click on the "Hello World!" string, then press the `<Delete>` key. You can now save your file (e.g., `<Ctrl>-<S>`).

Also, we no longer need the `hello_world` string resource. To remove it, double-click on the `res/values/strings.xml` file, select the `hello_world` string resource, click the "Remove..." button, click "Yes" on the confirmation dialog. Then you can save the file.

Step #2: Adding a ProgressBar

Now that the `TextView` is out of the way, we can add our `ProgressBar` in its place.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio

Go back to `res/layout/main.xml` in Android Studio. In the "Widgets" portion of the tool palette, you will see a few `ProgressBar` widget entries:

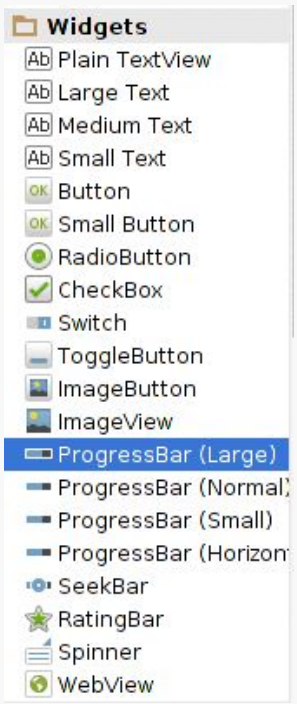


Figure 175: ProgressBar Widgets in the Tool Palette

Drag the "Large" one out of the palette and onto the preview of our activity. You will see a tooltip pointing out the `RelativeLayout` rules that the drag-and-drop operation will apply if you drop the widget in its current location. Slide the `ProgressBar` around until you center it and the tooltip shows that it will use `centerHorizontal` and `centerVertical` rules. If you wind up with `centerInParent` instead of those other two settings, that is fine as well.

In the Properties list on the right, find the "id" row, and change the value for the "id" to `progressBar1` (to match what Eclipse's drag-and-drop GUI builder uses, to keep the two sets of instructions synchronized).

At this point, you can save your file.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Eclipse

Go back to `res/layout/main.xml` in Eclipse. In the "Form Widgets" portion of the tool palette, you will see a few `ProgressBar` widget entries:

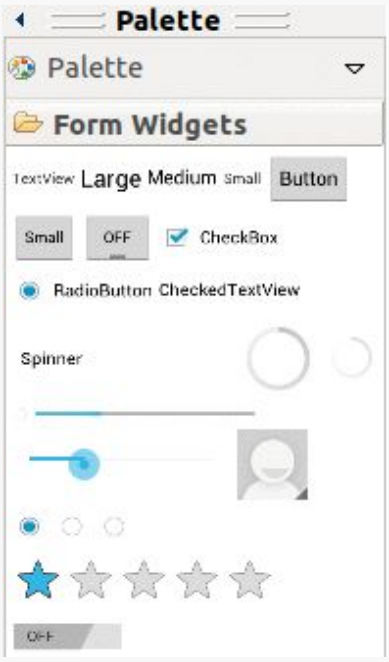


Figure 176: ProgressBar Widgets in the Tool Palette

Drag the "Large" one out of the palette and onto the preview of our activity. You will see a tooltip pointing out the `RelativeLayout` rules that the drag-and-drop operation will apply if you drop the widget in its current location. Slide the `ProgressBar` around until you center it and the tooltip shows that it will use `android:layout_centerHorizontal="true"` and `android:layout_centerVertical="true"`. If you wind up with `android:layout_centerInParent="true"` instead of those other two settings, that is fine as well.

If you are having difficulty centering it, drop it anywhere in the white part of the preview area. Then, from the toolbar above the preview, press the center-horizontal and center-vertical toolbar buttons in succession:

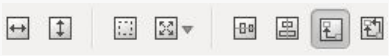


Figure 177: The Centering Toolbar Buttons (Third and Fourth from Right)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Then, you can save your file.

Step #3: Seeing the Results

If you run the app in a device or emulator, you will see your `ProgressBar` widget, sitting there, all alone, waiting for somebody to write more code in support of it:

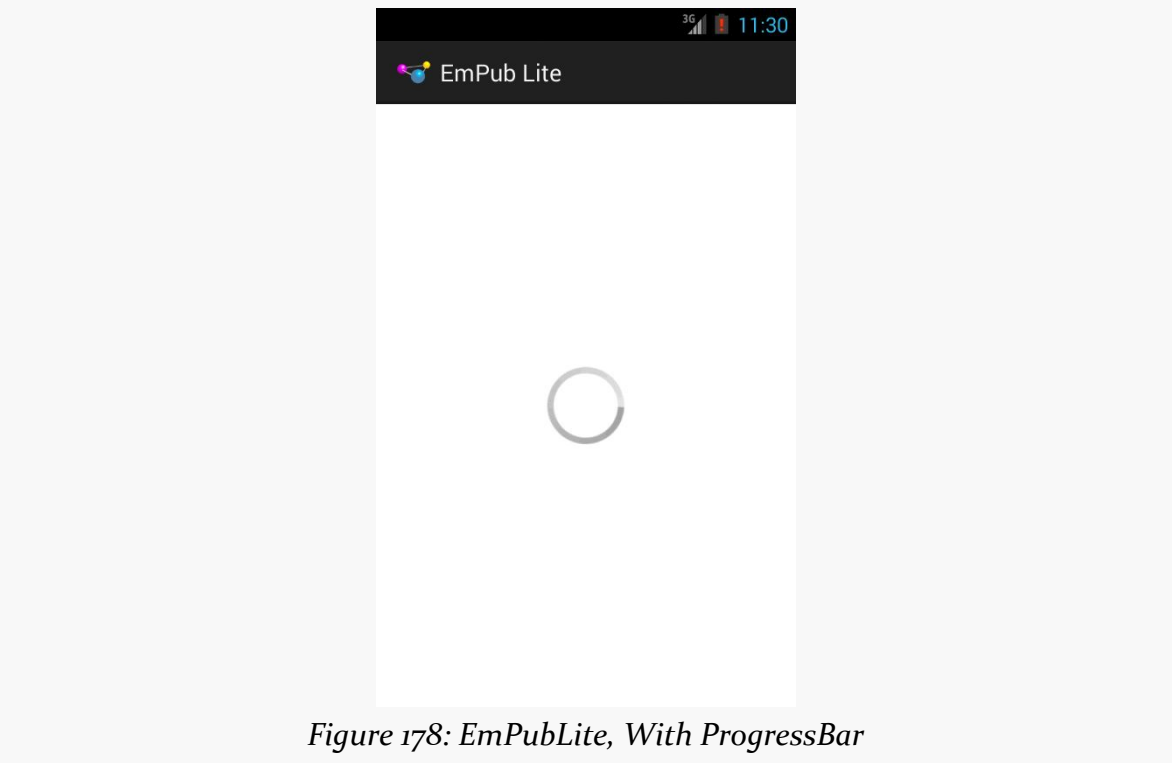


Figure 178: EmPubLite, With ProgressBar

Note that if you have not yet set up the x86 emulator, you might wish to consider doing so, as we will become increasingly dependent upon the emulator, and the ARM emulator is slow. There is a section later in the book that covers how to set up the x86 emulator.

In Our Next Episode…

… we will attach a third-party library to our tutorial project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

GUI Building, Continued

If you are using an IDE, and you have been experimenting with the graphical layout editor and drag-and-drop GUI building, this chapter will cover some other general features of this editor that you may find useful.

Even if you are not using an IDE, you may want to at least skim this chapter, as you will find a few tricks that will be relevant for you as well.

Making Your Selection

Clicking on a widget makes it the selected widget, meaning that the toolbar buttons will affect that widget (or, sometimes, its container, depending upon the button). Selected widgets have a thin blue border with blue square "grab handles" for adjusting its size and position.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 179: Android Studio, Selected Widget in Graphical Layout Editor

Clicking on a container also selects it. However, there may or may not be a blue border — in particular, containers that fill the screen (`match_parent` for width and height) do not seem to get the border.

Sometimes, though, you want to select a container that you cannot reach, because its contents are completely filled with widgets. That occurs with the `LinearPercent` sample from a previous chapter – the entire `LinearLayout` is filled with the three `Button` widgets. In these cases, click on the widget or container in the Component Tree (Android Studio) or Outline (Eclipse) pane to select it.

Including Includes

Sometimes, you have a widget or a collection of widgets that you want to reuse across multiple layout XML resources. Android supports the notion of an "include" that allows this. Simply create a dedicated layout XML resource that contains the widget(s) to reuse, then add them to your main layouts via an `<include>` element:

```
<include layout="@layout/thing_we_are_reusing" />
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You can even assign the `<include>` element a width or height if needed, as if it were just a widget or container.

The IDE makes it easy for you to take widgets from an existing layout XML resource and extract them into a separate layout XML resource, replacing them with an `<include>` element.

In Android Studio, select the widget(s) that you want to reuse, then choose Refactor > Extract Layout from the context menu. This will display a dialog where you can fill in the file name of your resulting resource:

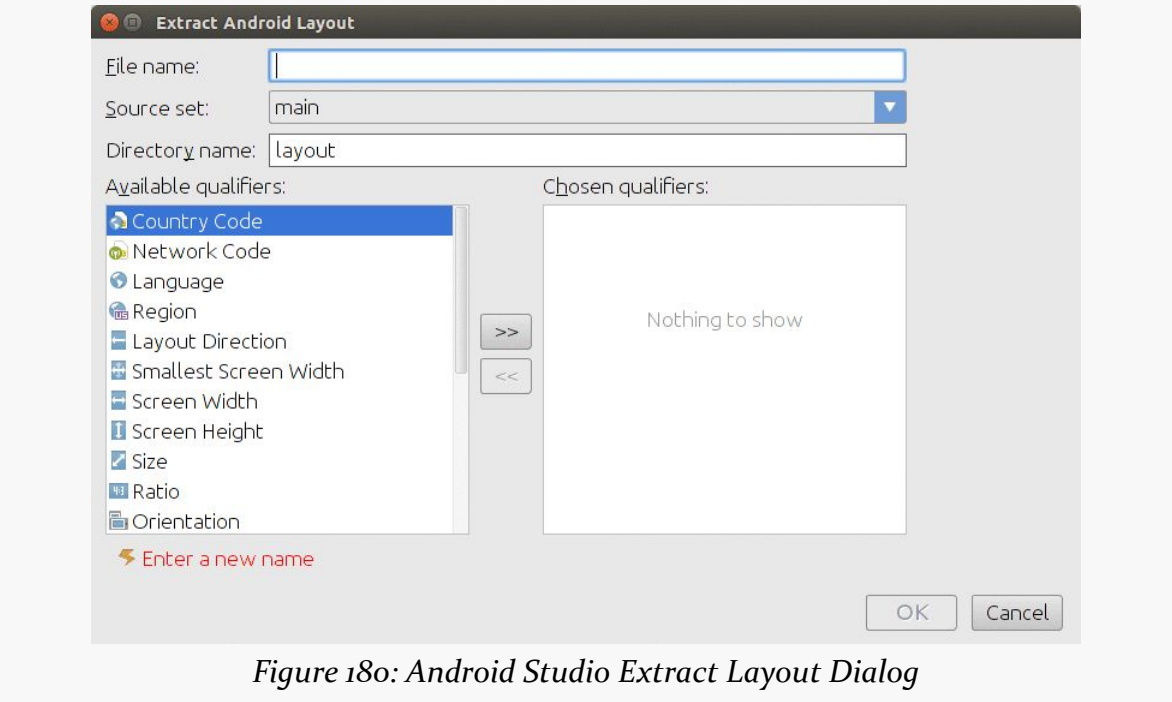


Figure 180: Android Studio Extract Layout Dialog

In Eclipse, select the widget(s) that you want to reuse, then right-click over them and choose "Extract Include" from the context menu. This will bring up a dialog where you can specify a name to give the new layout XML resource:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

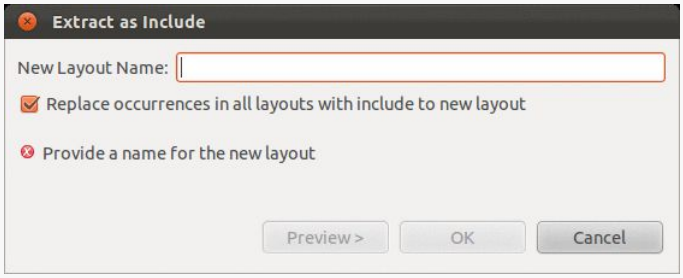


Figure 181: Extract as Include Dialog

By default, Eclipse will search *all* your layout files for these widgets and replace them with the `<include>`, though you can uncheck the checkbox to disable this behavior and only affect the layout XML resource you are presently editing.

If you are extracting multiple widgets that are not wrapped in their own container, the IDE will automatically wrap them in a `<merge>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android">
  <!-- widgets go here -->
</merge>
```

This is necessary purely from an XML standpoint — you cannot have multiple root elements in an XML file. When the `<merge>` is added to another layout via `<include>`, the `<merge>` element itself evaporates, leaving behind its children.

Wrap It Up (In a Container)

Sometimes, after you have added a widget to your layout, you later determine that you really needed it to be in some sort of container. For example, perhaps you thought you only needed one `TextView` but later decided to stack two `TextView` widgets in a vertical `LinearLayout`, in which case you somehow need to introduce this `LinearLayout` into the mix.

In Eclipse, the simplest way to do that is to right-click over the widget that needs a new container (in the preview pane or the Outline pane) and choose “Wrap In Container...” from the context menu. This will bring up a dialog allowing you to choose the class of the container (with a reasonable default pre-selected) and give the container an `android:id` value (which, for some strange reason, is mandatory).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

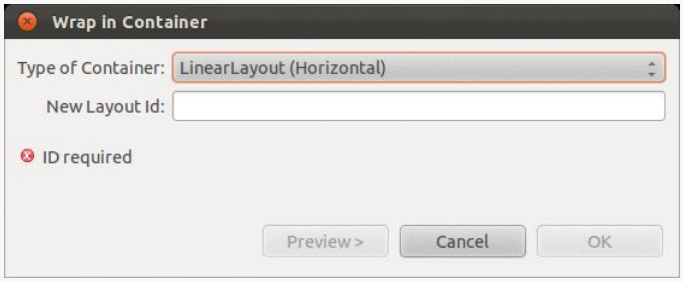


Figure 182: Wrap In Container Dialog

Similarly, if a widget is wrapped in a container, where the container is no longer necessary, "Remove Container" will get rid of the container.

Note that Android Studio does not offer this feature.

Morphing Widgets

Occasionally, you might configure a widget, only to decide later on that you really want it to be a different type of widget. For example, perhaps you start with a `CheckBox` and later want to switch it to be a `ToggleButton`.

To do this in Android Studio, right-click over the widget (in the preview pane or the Component Tree pane) and choose Morphing from the context menu. This brings up a fly-out menu of possible alternative classes; choosing one will automatically convert your widget into the selected type.

To do this in Eclipse, right-click over the widget (in the preview pane or the Outline pane) and choose "Change Widget Type" from the context menu. This will bring up a dialog box for you to choose a replacement widget class, with a likely candidate pre-selected for you:

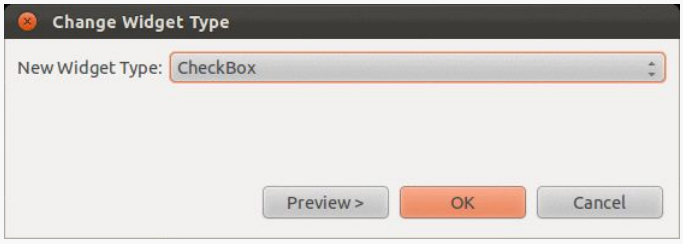


Figure 183: Change Widget Type

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

After making the selection, Eclipse will alter your element to the new widget type. Note that you may need to make other changes yourself, for attributes that you no longer need or now need to add.

Preview of Coming Attractions

At the top of the graphical layout editor, you will find a series of drop-downs that allow you to tailor what the preview looks like:

Figure 184: Android Studio Preview Controls

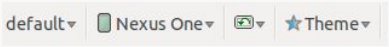


Figure 185: Eclipse Preview Controls

Your IDE will choose some likely defaults based upon your project settings, but you are welcome to change them as you see fit. Notable changes include:

- What version of Android is used for the preview (as widget styling changes from time to time in Android releases)
- What language is used for your string resources?
- What size and resolution of screen is used?
- Is it displayed in portrait or landscape?

These only affect the preview, so they show you (approximately) what your layout will look like under those conditions, but they do not modify anything about your layout XML itself.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

AdapterViews and Adapters

If you want the user to choose something out of a collection of somethings, you could use a bunch of `RadioButton` widgets. However, Android has a series of more flexible widgets than that, ones that this book will refer to as "selection widgets".

These include:

- `ListView`, which is your typical "list box"
- `Spinner`, which (more or less) is a drop-down list
- `GridView`, offering a two-dimensional roster of choices
- `ExpandableListView`, a limited "tree" widget, supporting two levels in the hierarchy

and many more.

At their core, these are ordinary widgets. You will find them in your tool palette of your IDE's graphical layout editor, and can drag them and position them as you see fit.

The key is that these all have a common superclass: `AdapterView`, so named because they partner with objects implementing the `Adapter` interface to determine what choices are available for the user to choose from.

Adapting to the Circumstances

An `Adapter` is your bridge between your model data and that data's visual representation in the `AdapterView`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- an Adapter might "adapt" an Invoice into a View that would serve as a row in a ListView
- an Adapter might "adapt" a Book into a View that would serve as a cell in a GridView
- and so on

Android ships with several Adapter classes ready for your use, where the different adapter classes are designed to "adapt" different sorts of collections (e.g., arrays versus results of database queries). Android also has a BaseAdapter class that can serve as the foundation for your own Adapter implementation, if you need to "adapt" a collection of data that does not fit any of the Adapter classes supplied by Android.

Using ArrayAdapter

The easiest adapter to use is ArrayAdapter — all you need to do is wrap one of these around a Java array or java.util.List instance, and you have a fully-functioning adapter:

```
String[] items={"this", "is", "a", "really", "silly", "list"};
new ArrayAdapter<String>(this,
                         android.R.layout.simple_list_item_1,
                         items);
```

One flavor of the ArrayAdapter constructor takes three parameters:

1. The Context to use (typically this will be your activity instance)
2. The resource ID of a view to use (such as a built-in system resource ID, as shown above)
3. The actual array or list of items to show

By default, the ArrayAdapter will invoke toString() on the objects in the list and wrap each of those strings in the view designated by the supplied resource. android.R.layout.simple_list_item_1 simply turns those strings into TextView objects. Those TextView widgets, in turn, will be shown in the list or spinner or whatever widget uses this ArrayAdapter. If you want to see what android.R.layout.simple_list_item_1 looks like, you can find a copy of it in your SDK installation — just search for simple_list_item_1.xml.

We will see in a later section how to subclass an Adapter and override row creation, to give you greater control over how rows and cells appear.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Lists of Naughty and Nice

The classic listbox widget in Android is known as `ListView`. Include one of these in your layout, invoke `setAdapter()` to supply your data and child views, and attach a listener via `setOnItemSelectedListener()` to find out when the selection has changed. With that, you have a fully-functioning listbox.

However, if your activity is dominated by a single list, you might well consider creating your activity as a subclass of `ListActivity`, rather than the regular `Activity` base class. If your main view is just the list, you do not even need to supply a layout — `ListActivity` will construct a full-screen list for you. If you do want to customize the layout, you can, so long as you identify your `ListView` as `@android:id/list`, so `ListActivity` knows which widget is the main list for the activity.

For example, here is a layout pulled from the `Selection/List` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent" >
  <TextView
    android:id="@+id/selection"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    />
</LinearLayout>
```

It is just a list with a label on top to show the current selection.

The Java code to configure the list and connect the list with the label is:

```
package com.commonsware.android.list;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public class ListViewDemo extends ListActivity {
  private TextView selection;
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    setListAdapter(new ArrayAdapter<String>(this,
                        android.R.layout.simple_list_item_1,
                        items));
    selection=(TextView)findViewById(R.id.selection);
  }

  @Override
  public void onListItemClick(ListView parent, View v, int position,
                              long id) {
    selection.setText(items[position]);
  }
}
```

With `ListActivity`, you can set the list adapter via `setListAdapter()` — in this case, providing an `ArrayAdapter` wrapping an array of Latin strings. To find out when the list selection changes, override `onListItemClick()` and take appropriate steps based on the supplied child view and position (in this case, updating the label with the text for that position).

The results?

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

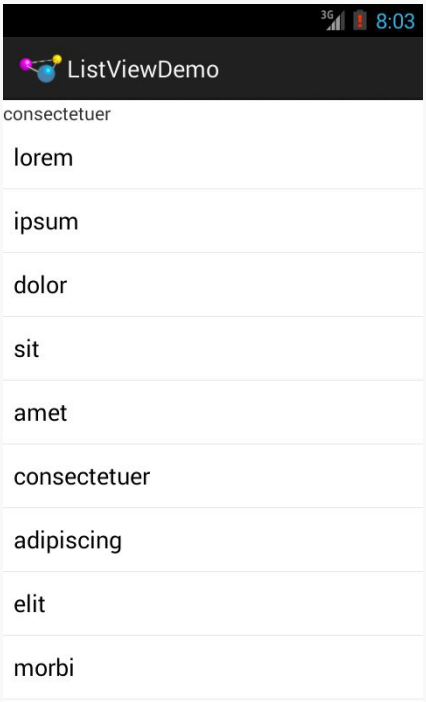


Figure 186: ListViewDemo, After User Taps on "consecteteur"

The second parameter to our `ArrayAdapter` — `android.R.layout.simple_list_item_1` — controls what the rows look like. The value used in the preceding example provides the standard Android list row: a big font with lots of padding to offer a large touch target for the user.

Clicks versus Selections

One thing that can confuse some Android developers is the distinction between clicks and selections. One might think that they are the same thing — after all, clicking on something selects it, right?

Well, no. At least, not in Android. At least not all of the time.

Android is designed to be used with touchscreen devices and non-touchscreen devices. Historically, Android has been dominated by devices that only offered touchscreens. However, there are various devices powered by Android and connected to TVs. Most TVs are not touchscreens, and so users of those TV-using Android devices will use some sort of remote control to drive Android. And some

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android devices offer both touchscreens and some other sort of pointing device — D-pad, trackball, arrow keys, etc.

To accommodate both styles of device, Android sometimes makes a distinction between selection events and click events. Widgets based off of the "spinner" paradigm — including `Spinner` — treat everything as selection events. Other widgets — like `ListView` and `GridView` — treat selection events and click events differently. For these widgets, selection events are driven by the pointing device, such as using arrow keys to move a highlight bar up and down a list. Click events are when the user either "clicks" the pointing device (e.g., presses the center D-pad button) *or* taps on something in the widget using the touchscreen.

Choice Modes

By default, `ListView` is set up simply to collect clicks on list entries. Sometimes, though, you want a list that tracks a user's choice, or possibly multiple choices. `ListView` can handle that as well, but it requires a few changes.

First, you will need to call `setChoiceMode()` on the `ListView` in Java code to set the choice mode, classically supplying either `CHOICE_MODE_SINGLE` or `CHOICE_MODE_MULTIPLE` as the value. You can get your `ListView` from a `ListActivity` via `getListView()`. You can also declare this via the `android:choiceMode` attribute in your layout XML.

Then, rather than use `android.R.layout.simple_list_item_1` as the layout for the list rows in your `ArrayAdapter` constructor, you can use either `android.R.layout.simple_list_item_single_choice` or `android.R.layout.simple_list_item_multiple_choice` for single-choice or multiple-choice lists, respectively.

For example, here is an activity layout from the `Selection/Checklist` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<ListView
xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/list"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:drawSelectorOnTop="false"
  android:choiceMode="multipleChoice"
/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It is a full-screen `ListView`, with the `android:choiceMode="multipleChoice"` attribute to indicate that we want multiple choice support.

Our activity just uses a standard `ArrayAdapter` on our list of Latin words, but uses `android.R.layout.simple_list_item_multiple_choice` as the row layout:

```
package com.commonsware.android.checklist;

import android.app.ListActivity;
import android.os.Bundle;
import android.widget.ArrayAdapter;

public class ChecklistDemo extends ListActivity {
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    setListAdapter(new ArrayAdapter<String>(this,
                        android.R.layout.simple_list_item_multiple_choice,
                        items));
  }
}
```

What the user sees is the list of words with checkboxes down the right edge:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

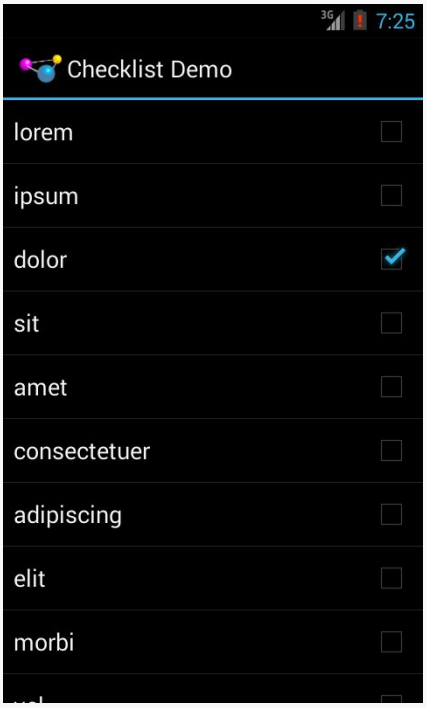


Figure 187: Multiple-Choice Mode

If we wanted, we could call methods like `getCheckedItemPositions()` on our `ListView` to find out which items the user checked, or `setItemChecked()` if we wanted to check (or un-check) a specific entry ourselves.

Clicks versus Selections, Revisited

If the user clicks a row in a `ListView`, a click event is registered, triggering things like `onListItemClick()` in an `OnItemClickListener`. If the user uses a pointing device to change a selection (e.g., pressing up and down arrows to move a highlight bar in the `ListView`), that triggers `onItemSelected()` in an `OnItemSelectedListener`.

Many times, particularly if the `ListView` is the entire UI at present, you only care about clicks. Sometimes, particularly if the `ListView` is adjacent to something else (e.g., on a TV, where you have more screen space *and* do not have a touchscreen), you will care more about selection events. Either way, you can get the events you need.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Spin Control

In Android, the `Spinner` is the equivalent of the drop-down selector you might find in other toolkits. Clicking the `Spinner` drops down a list for the user to choose an item from. You basically get the ability to choose an item from a list without taking up all the screen space of a `ListView`, at the cost of an extra click to make a change.

As with `ListView`, you provide the adapter for data and child views via `setAdapter()` and hook in a listener object for selections via `setOnItemSelectedListener()`.

To tailor the view used when displaying the drop-down perspective, you need to configure the adapter, not the `Spinner` widget. Use the `setDropDownViewResource()` method to supply the resource ID of the view to use.

For example, culled from the `Selection/Spinner` sample project, here is an XML layout for a simple view with a `Spinner`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TextView
    android:id="@+id/selection"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    />
  <Spinner android:id="@+id/spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
  />
</LinearLayout>
```

This is the same view as shown in a previous section, just with a `Spinner` instead of a `ListView`.

To populate and use the `Spinner`, we need some Java code:

```
public class SpinnerDemo extends Activity
  implements AdapterView.OnItemSelectedListener {
  private TextView selection;
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    selection=(TextView)findViewById(R.id.selection);

    Spinner spin=(Spinner)findViewById(R.id.spinner);
    spin.setOnItemSelectedListener(this);

    ArrayAdapter<String> aa=new ArrayAdapter<String>(this,
                              android.R.layout.simple_spinner_item,
                              items);

    aa.setDropDownViewResource(
      android.R.layout.simple_spinner_dropdown_item);
    spin.setAdapter(aa);
  }

  @Override
  public void onItemSelected(AdapterView<?> parent,
                             View v, int position, long id) {
    selection.setText(items[position]);
  }

  @Override
  public void onNothingSelected(AdapterView<?> parent) {
    selection.setText("");
  }
}
```

Here, we attach the activity itself as the selection listener (`spin.setOnItemSelectedListener(this)`), as `Spinner` widgets only support selection events, not click events. This works because the activity implements the `OnItemSelectedListener` interface. We configure the adapter not only with the list of fake words, but also with a specific resource to use for the drop-down view (via `aa.setDropDownViewResource()`). Also note the use of `android.R.layout.simple_spinner_item` as the built-in `View` for showing items in the spinner itself. Finally, we implement the callbacks required by `OnItemSelectedListener` to adjust the selection label based on user input.

What we get is:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

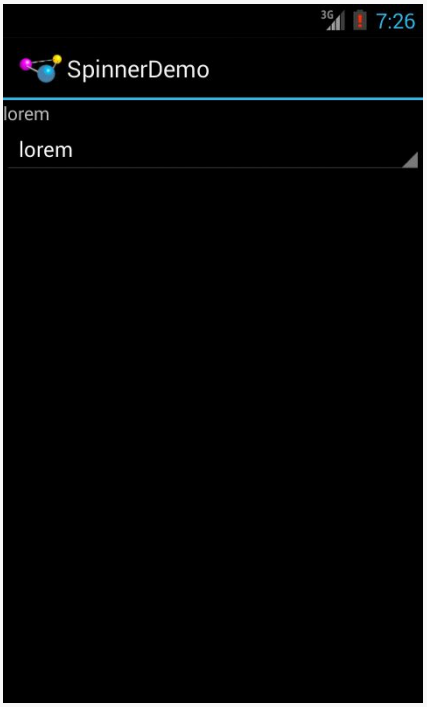


Figure 188: SpinnerDemo, as Initially Launched

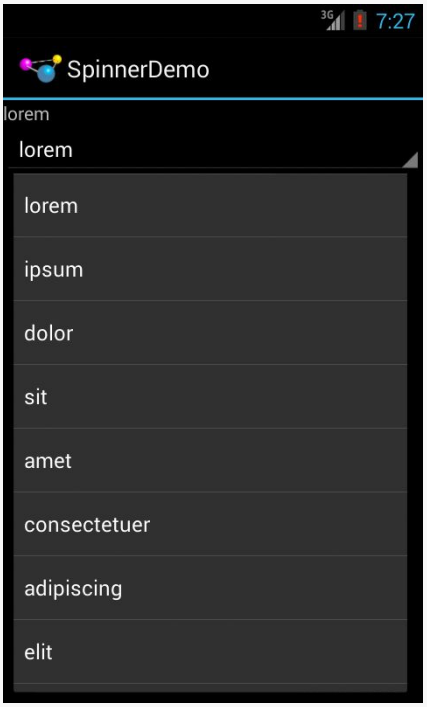


Figure 189: SpinnerDemo, with Spinner Drop-Down List Displayed

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Grid Your Lions (Or Something Like That…)

As the name suggests, `GridView` gives you a two-dimensional grid of items to choose from. You have moderate control over the number and size of the columns; the number of rows is dynamically determined based on the number of items the supplied adapter says are available for viewing.

There are a few properties which, when combined, determine the number of columns and their sizes:

1. `android:numColumns` spells out how many columns there are, or, if you supply a value of `auto_fit`, Android will compute the number of columns based on available space and the properties listed below.
2. `android:verticalSpacing` and `android:horizontalSpacing` indicate how much whitespace there should be between items in the grid.
3. `android:columnWidth` indicates how wide each column should be, in terms of some dimension value (e.g., `40dp` or `@dimen/grid_column_width`).
4. `android:stretchMode` indicates, for grids with `auto_fit` for `android:numColumns`, what should happen for any available space not taken up by columns or spacing — this should be `columnWidth` to have the columns take up available space or `spacingWidth` to have the whitespace between columns absorb extra space.

Otherwise, the `GridView` works much like any other selection widget — use `setAdapter()` to provide the data and child views, invoke `setOnItemClickListener()` to find out when somebody clicks on a cell in the grid, etc.

For example, here is an XML layout from the `Selection/Grid` sample project, showing a `GridView` configuration:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TextView
    android:id="@+id/selection"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    />
  <GridView
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:id="@+id/grid"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:verticalSpacing="40dip"
    android:horizontalSpacing="5dip"
    android:numColumns="auto_fit"
    android:columnWidth="100dip"
    android:stretchMode="columnWidth"
    android:gravity="center"
    />
</LinearLayout>
```

For this grid, we take up the entire screen except for what our selection label requires. The number of columns is computed by Android (`android:numColumns = "auto_fit"`) based on our horizontal spacing (`android:horizontalSpacing = "5dip"`) and columns width (`android:columnWidth = "100dip"`), with the columns absorbing any "slop" width left over (`android:stretchMode = "columnWidth"`).

The Java code to configure the `GridView` is:

```
package com.commonsware.android.grid;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.GridView;
import android.widget.TextView;

public class GridDemo extends Activity
  implements AdapterView.OnItemClickListener {
  private TextView selection;
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    selection=(TextView)findViewById(R.id.selection);

    GridView g=(GridView) findViewById(R.id.grid);
    g.setAdapter(new ArrayAdapter<String>(this,
                        R.layout.cell,
                        items));
    g.setOnItemClickListener(this);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }

  @Override
  public void onItemClick(AdapterView<?> parent, View v,
                          int position, long id) {
    selection.setText(items[position]);
  }
}
```

The grid cells are defined by a separate `res/layout/cell.xml` file, referenced in our `ArrayAdapter` as `R.layout.cell`:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:textSize="14dip"
/>
```

With the vertical spacing from the XML layout (`android:verticalSpacing = "40dip"`), the grid overflows the boundaries of the emulator's screen:

Figure 190: GridDemo, as Initially Launched

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 191: GridDemo, Scrolled to the Bottom of the Grid

`GridView`, like `ListView`, supports both click events and selection events. In this sample, we register an `OnItemClickListener` to listen for click events.

Fields: Now With 35% Less Typing!

The `AutoCompleteTextView` is sort of a hybrid between the `EditText` (field) and the `Spinner`. With auto-completion, as the user types, the text is treated as a prefix filter, comparing the entered text as a prefix against a list of candidates. Matches are shown in a selection list that folds down from the field. The user can either type out an entry (e.g., something not in the list) or choose an entry from the list to be the value of the field.

`AutoCompleteTextView` subclasses `EditText`, so you can configure all the standard look-and-feel aspects, such as font face and color.

In addition, `AutoCompleteTextView` has an `android:completionThreshold` property, to indicate the minimum number of characters a user must enter before the list filtering begins.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You can give `AutoCompleteTextView` an adapter containing the list of candidate values via `setAdapter()`. However, since the user could type something not in the list, `AutoCompleteTextView` does not support selection listeners. Instead, you can register a `TextWatcher`, like you can with any `EditText`, to be notified when the text changes. These events will occur either because of manual typing or from a selection from the drop-down list.

Below we have a familiar-looking XML layout, this time containing an `AutoCompleteTextView` (pulled from the `Selection/AutoComplete` sample application):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TextView
    android:id="@+id/selection"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    />
  <AutoCompleteTextView android:id="@+id/edit"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:completionThreshold="3"/>
</LinearLayout>
```

The corresponding Java code is:

```
package com.commonsware.android.auto;

import android.app.Activity;
import android.os.Bundle;
import android.text.Editable;
import android.text.TextWatcher;
import android.widget.ArrayAdapter;
import android.widget.AutoCompleteTextView;
import android.widget.TextView;

public class AutoCompleteDemo extends Activity
  implements TextWatcher {
  private TextView selection;
  private AutoCompleteTextView edit;
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    selection=(TextView)findViewById(R.id.selection);
    edit=(AutoCompleteTextView)findViewById(R.id.edit);
    edit.addTextChangedListener(this);

    edit.setAdapter(new ArrayAdapter<String>(this,
                            android.R.layout.simple_dropdown_item_1line,
                            items));
  }

  @Override
  public void onTextChanged(CharSequence s, int start, int before,
                            int count) {
    selection.setText(edit.getText());
  }

  @Override
  public void beforeTextChanged(CharSequence s, int start,
                                int count, int after) {
    // needed for interface, but not used
  }

  @Override
  public void afterTextChanged(Editable s) {
    // needed for interface, but not used
  }
}
```

This time, our activity implements `TextWatcher`, which means our callbacks are `onTextChanged()`, `beforeTextChanged()`, and `afterTextChanged()`. In this case, we are only interested in the first, and we update the selection label to match the `AutoCompleteTextView`'s current contents.

Here we have the results:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

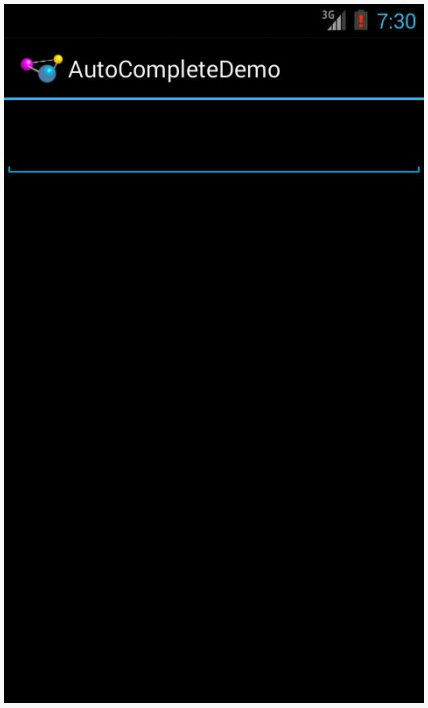


Figure 192: AutoCompleteDemo, as Initially Launched

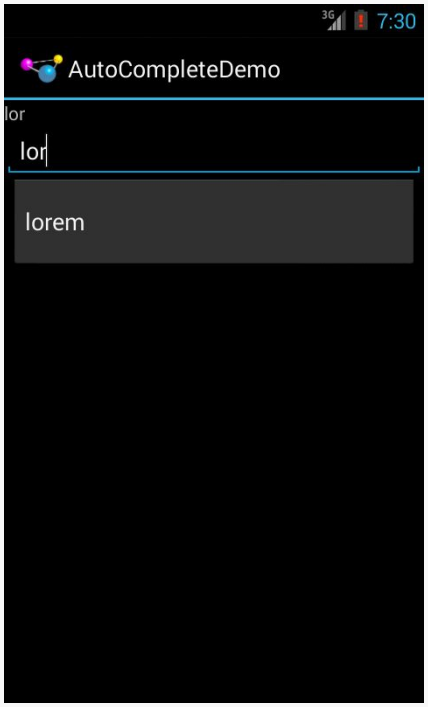


Figure 193: AutoCompleteDemo, After Entering a Few Matching Letters

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

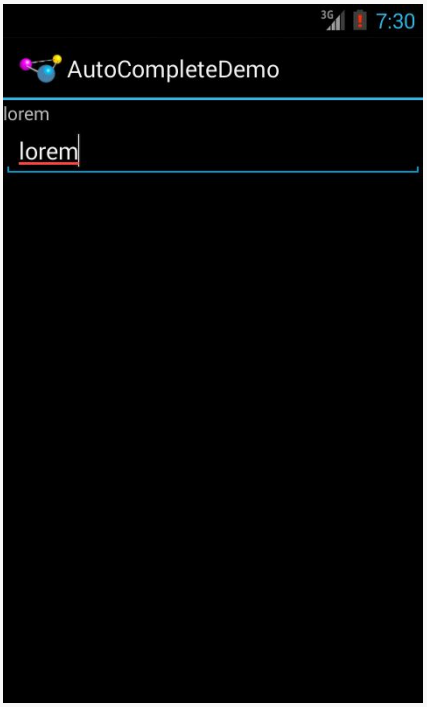


Figure 194: AutoCompleteDemo, After Auto-Complete Value Was Selected

Note that the red underline in the preceding screenshot is due to spelling correction. Like `EditText`, `AutoCompleteTextView` supports hinting at spelling errors. The emulator's language is set to English, as there is no option in it for Latin.

Customizing the Adapter

The humble `ListView` is one of the most important widgets in all of Android, simply because it is used so frequently. Whether choosing a contact to call or an email message to forward or an ebook to read, `ListView` widgets are employed in a wide range of activities.

Of course, it would be nice if they were more than just plain text.

The good news is that they can be as fancy as you want, within the limitations of a mobile device's screen, of course. However, making them more elaborate takes some work.

Note that while this section will be using `ListView` as the `AdapterView`, the same techniques hold for *any* `AdapterView`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Single Layout Pattern

The simplest way of creating custom `ListView` rows (or `GridView` cells or whatever) is when they all have the same basic structure and can be created from the same layout XML resource. This does not mean they have to be strictly identical, but that you can make whatever changes you need just by configuring the widgets (e.g., make some things `VISIBLE` or `GONE`).

This is not especially difficult, though it does take a few more steps than what we have seen previously.

Step #0: Get Things Set Up Simply

First, create your activity (e.g., `ListActivity`), get your data (e.g., array of Java strings), and set up your `AdapterView` with a simple adapter following the steps outlined in the preceding sections.

Here, we will examine the `Selection/Dynamic` sample project. We will use a simple `ListActivity` (taking the default layout of a full-screen `ListView`) and use the same list of 25 Latin words used in earlier samples. However, this time, we want to have a more elaborate row, taking into account the length of the Latin word.

Step #1: Design Your Row

Next, create a layout XML resource that will represent one row in your `ListView` (or cell in your `GridView` or whatever).

For example, our `res/layout/row.xml` resource will use a pair of nested `LinearLayout` containers to organize two `TextView` widgets and an `ImageView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal">

  <ImageView
    android:id="@+id/icon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:padding="2dip"
    android:src="@drawable/ok"
    android:contentDescription="@string/icon"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
      android:id="@+id/label"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="25sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/size"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="15sp"/>
  </LinearLayout>

</LinearLayout>
```

The `ImageView` will use one of two drawable resources, one for short words, and another for long words.

Step #2: Extend ArrayAdapter

If you just used `R.layout.row` with a regular `ArrayAdapter`, it would work, insofar as it would not crash. However, `ArrayAdapter` only knows how to update a single `TextView` in a row, so it would ignore our other `TextView`, let alone the `ImageView`.

So, we need to create our own `ListAdapter`, by creating our own subclass of `ArrayAdapter`.

Since an `Adapter` is tightly coupled to the `AdapterView` that uses it, it is typically simplest to make the custom `ArrayAdapter` subclass be an inner class of whoever manages the `AdapterView`. Hence, in our sample, we will create an `IconicAdapter` inner class of our `ListActivity`.

Step #3: Override the Constructor and `getView()`

The `IconicAdapter` constructor can chain to the superclass and supply the necessary data, such as our Java array of Latin words. The real fun comes when we override `getView()`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.fancylists.three;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
import android.widget.TextView;

public class DynamicDemo extends ListActivity {
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setListAdapter(new IconicAdapter());
  }

  class IconicAdapter extends ArrayAdapter<String> {
    IconicAdapter() {
      super(DynamicDemo.this, R.layout.row, R.id.label, items);
    }

    @Override
    public View getView(int position, View convertView,
                        ViewGroup parent) {
      View row=super.getView(position, convertView, parent);
      ImageView icon=(ImageView)row.findViewById(R.id.icon);

      if (items[position].length()>4) {
        icon.setImageResource(R.drawable.delete);
      }
      else {
        icon.setImageResource(R.drawable.ok);
      }

      TextView size=(TextView)row.findViewById(R.id.size);

      size.setText(String.format(getString(R.string.size_template),
items[position].length()));

      return(row);
    }
  }
}
```

Our `getView()` implementation does three things:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- It chains to the superclass' implementation of `getView()`, which returns to us an instance of our row `View`, as prepared by `ArrayAdapter`. In particular, our word has already been put into one `TextView`, since `ArrayAdapter` does that normally.
- It finds our `ImageView` and applies a business rule to set which icon should be used, referencing one of two drawable resources (`R.drawable.ok` and `R.drawable.delete`).
- It finds our other `TextView` and populates it as well, by pulling in the value of a string resource and using `String.format()` to pour in our word length.

Note that we call `findViewById()` not on the activity, but rather on the row returned by the superclass' implementation of `getView()`. **Always call `findViewById()` on something that is guaranteed to give you a unique result.** In the case of an `AdapterView`, there will be many rows, cells, etc. — calling `findViewById()` on the activity might return widgets with the right name but from other rows or cells.

This gives us:

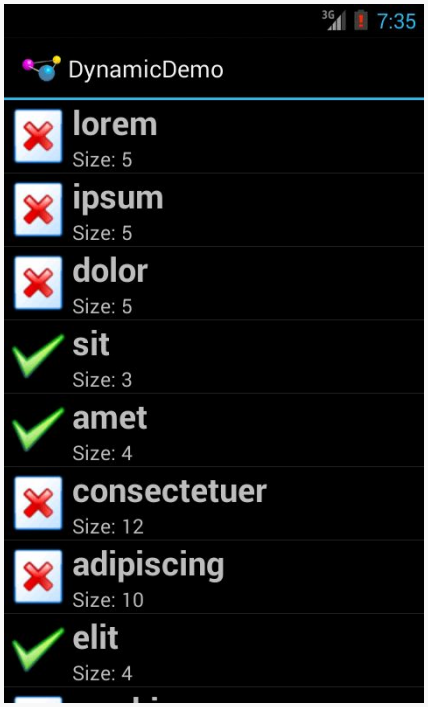


Figure 195: The Dynamic Sample Application

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The approach of overriding `getView()` works for `ArrayAdapter`, but some other types of adapters would have alternatives. We will see that mostly with `CursorAdapter`, profiled in upcoming chapters.

Optimizing with the ViewHolder Pattern

A somewhat expensive operation we do a lot with more elaborate list rows is call `findViewById()`. This dives into our row and pulls out widgets by their assigned identifiers, so we can customize the widget contents (e.g., change the text of a `TextView`, change the icon in an `ImageView`). Since `findViewById()` can find widgets anywhere in the tree of children of the row's root `View`, this could take a fair number of instructions to execute, particularly if we keep having to re-find widgets we had found once before.

In some GUI toolkits, this problem is avoided by having the composite `View` objects, like our rows, be declared totally in program code (in this case, Java). Then, accessing individual widgets is merely the matter of calling a getter or accessing a field. And you can certainly do that with Android, but the code gets rather verbose. What would be nice is a way where we can still use the layout XML yet cache our row's key child widgets so we only have to find them once.

That's where the holder pattern comes into play, in a class we will call `ViewHolder`.

All `View` objects have `getTag()` and `setTag()` methods. These allow you to associate an arbitrary object with the widget. What the holder pattern does is use that "tag" to hold an object that, in turn, holds each of the child widgets of interest. By attaching that holder to the row `View`, every time we use the row, we already have access to the child widgets we care about, without having to call `findViewById()` again.

So, let's take a look at one of these holder classes (taken from the `Selection/ViewHolder` sample project, a revised version of the `Selection/Dynamic` sample from before):

```
package com.commonsware.android.fancylists.five;

import android.view.View;
import android.widget.ImageView;
import android.widget.TextView;

class ViewHolder {
  ImageView icon=null;
  TextView size=null;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  ViewHolder(View row) {
    this.icon=(ImageView)row.findViewById(R.id.icon);
    this.size=(TextView)row.findViewById(R.id.size);
  }
}
```

`ViewHolder` holds onto the child widgets, initialized via `findViewById()` in its constructor. The widgets are simply package-protected data members, accessible from other classes in this project... such as a `ViewHolderDemo` activity. In this case, we are only holding onto two widgets — the icon and the second label – since we will let `ArrayAdapter` handle our first label for us. In our case, we are holding onto the `TextView` and `ImageView` widgets that we want to populate in `getView()`.

Using `ViewHolder` is a matter of creating an instance whenever we inflate a row and attaching said instance to the row `View` via `setTag()`, as shown in this rewrite of `getView()`, found in `ViewHolderDemo`:

```
    @Override
    public View getView(int position, View convertView,
                        ViewGroup parent) {
      View row=super.getView(position, convertView, parent);
      ViewHolder holder=(ViewHolder)row.getTag();

      if (holder==null) {
        holder=new ViewHolder(row);
        row.setTag(holder);
      }

      if (getModel(position).length()>4) {
        holder.icon.setImageResource(R.drawable.delete);
      }
      else {
        holder.icon.setImageResource(R.drawable.ok);
      }

      holder.size.setText(String.format(getString(R.string.size_template),
items[position].length()));

      return(row);
    }
```

If the call to `getTag()` on the row returns `null`, we know we need to create a new `ViewHolder`, which we then attach to the row via `setTag()` for later reuse. Then, accessing the child widgets is merely a matter of accessing the data members on the holder.

This takes advantage of the fact that rows in a `ListView` get *recycled* – a 25,000-row list does not create 25,000 rows. The recycling itself is handled for us by

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`ArrayAdapter`, so we simply have to create our `ViewHolder` when needed and reuse the existing `ViewHolder` when a row gets recycled. The first time the `ListView` is displayed, all new rows need to be created, and we wind up creating a `ViewHolder` for each. As the user scrolls, rows get recycled, and we can reuse their corresponding `ViewHolder` widget caches. We will cover this recycling process in greater detail in a later chapter.

Note that the `getModel()` method shown here retrieves our model `String` for a given `position`, by using `getListAdapter()` (to retrieve our `IconicAdapter` from the activity's `ListView`) and `getItem()` (to retrieve the data, held by the adapter, represented by the position):

```
  private String getModel(int position) {
    return(((IconicAdapter)getListAdapter()).getItem(position));
  }
```

Dealing with Multiple Row Layouts

The story gets significantly more complicated if our mix of rows is more complicated. For example, here is the Sound screen in the Settings application:

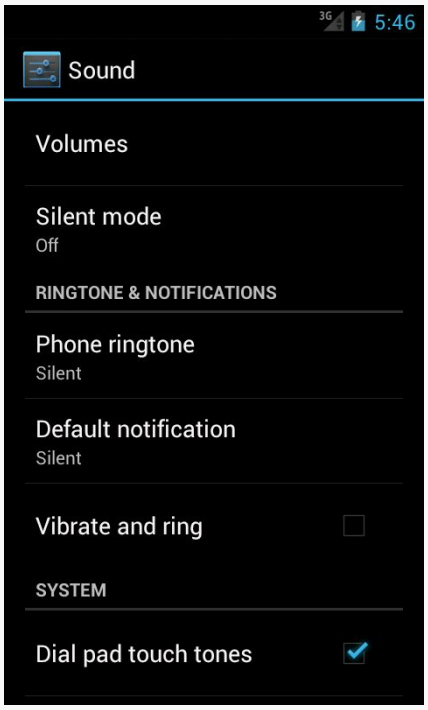


Figure 196: Sound Settings Screen

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It may not look like it, but that is a `ListView`. However, not all the rows look the same:

- Some have one line of text (e.g., “Volumes”)
- Some have two lines of text (e.g., “Silent mode” plus “Off”)
- Some have one line of text and a `CheckBox` (e.g., “Vibrate and ring”)
- Some are headings with totally different text formatting (e.g., “RINGTONE & NOTIFICATIONS”)

This is handled by having more than one row layout XML resource used by the adapter. The complexity comes not only in managing those different resources and determining which to use when, but in just having more than one resource – after all, we only teach `ArrayAdapter` how to use one. We will examine how to handle this scenario in a later chapter.

Visit the Trails!

To learn more about `ListView`, you can turn to Advanced ListViews, which covers other tricks you can do with a `ListView`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The WebView Widget

HTML has come a *long* way from Sir Tim Berners-Lee's original vision of using it to publish physics papers.

Not surprisingly, displaying HTML, CSS, and JavaScript in mobile applications is fairly popular, not only for creating full-fledged Web browsers, but for rendering HTML content from RSS/Atom feeds, from HTML-formatted email messages, ebooks (like the one you are reading), and so forth.

There are a couple of ways to display HTML in Android, with the most powerful being the `WebView` widget, the focus of this chapter.

Role of WebView

If your HTML is fairly limited in scope, such as what you might find in the body of a status update on Twitter, you can use the static `fromHtml()` method on the `Html` utility class to parse an HTML-formatted string into something that you can put into a `TextView`. `TextView` can render simple formatting like styles (bold, italic, etc.), font faces (serif, sans serif, etc.), colors, links, and so forth.

However, sometimes your needs for HTML transcend what `TextView` can handle. You will not be browsing Facebook using `TextView`, for example.

In those cases, `WebView` will be the more appropriate widget, as it can handle a *much* wider range of HTML tags. `WebView` can also handle CSS and JavaScript, which `Html.fromHtml()` would simply ignore. `WebView` can also assist you with common "browsing" metaphors, such as history list of visited URLs to support backwards and forwards navigation.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

On the other hand, `WebView` is a much more expensive widget to use, in terms of memory consumption, than is `TextView`.

WebView and WebKit

The reason for the memory cost of `WebView` is the fact that `WebView` is powered by a fairly complete copy of WebKit (for Android 4.3 and older) or Blink (for Android 4.4+). WebKit is an open source Web rendering engine that forms the heart of major Web browsers, such as Safari, while Blink is used for Google's Chrome and Chromium browsers. While the version of WebKit/Blink that lives in Android is one optimized for mobile use, it still represents a fairly substantial code base, and rendering complex Web pages takes up a fair amount of RAM (as anyone with lots of browser tabs on their desktop knows all too well).

Because `WebView` is powered by WebKit/Blink, content that renders in Chrome and Safari *probably* renders the same in `WebView`. The emphasis on the word "probably" is for a few reasons:

- As mentioned, WebKit/Blink in Android is a mobile-optimized version, which introduces some differences compared to its desktop brethren
- WebKit/Blink, like any software project, has its own upgrade cycles and versioning, so different browsers (Chrome vs. Safari vs. `WebView`) will use different versions of the WebKit engine, introducing some differences
- Android has tweaked WebKit/Blink for its own purposes, introducing yet other potential differences

Adding the Widget

For simple stuff, `WebView` is not significantly different than any other widget in Android — pop it into a layout, tell it what URL to navigate to via Java code, and you are done.

As you can see in the `WebKit/Browser1` sample application, here is a simple layout with a `WebView`:

```
<?xml version="1.0" encoding="utf-8"?>
<WebView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/webkit"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

As with any other widget, you need to tell it how it should fill up the space in the layout (in this case, it fills all remaining space).

And, just as with other widgets, you can add it using your IDE's graphical layout editor. An Android Studio user can drag a `WebView` out of the "Layouts" section of the tool palette, while an Eclipse user can drag a `WebView` out of the "Composite" section of the Eclipse tool palette.

Note that `WebView` knows how to scroll its own contents, so you do not need to put it in a `ScrollView` or `HorizontalScrollView`.

Loading Content Via a URL

There are a number of ways to load HTML content into a `WebView` widget.

The simplest is to use the `loadUrl()` method, which takes a URL and retrieves its contents over the Internet. For example, here is the activity source code for the `WebKit/Browser1` sample application:

```
package com.commonsware.android.browser1;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebView;

public class BrowserDemo1 extends Activity {
  WebView browser;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);

    browser.loadUrl("http://commonsware.com");
  }
}
```

However, we also have to make one change to `AndroidManifest.xml`, adding a line where we request permission to access the Internet:

```
  <uses-permission android:name="android.permission.INTERNET"/>
```

If we fail to add this permission, the browser will refuse to load pages. We will discuss more about this "permission" concept in a later chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The resulting activity looks like a Web browser, just with hidden scrollbars:

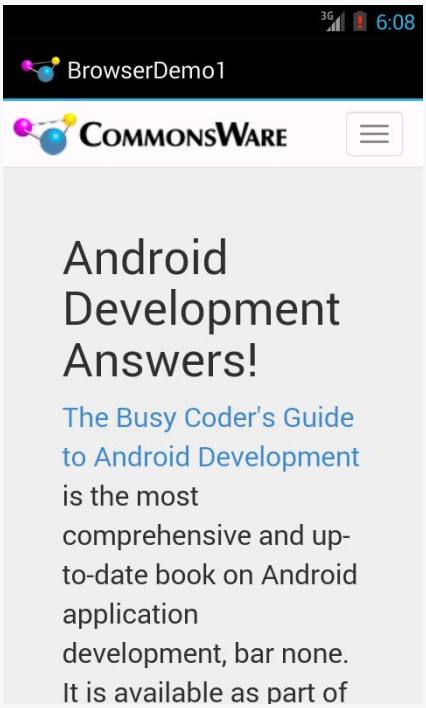


Figure 197: The Browser1 Sample Application (image from July 2012)

As with a regular Android Web browser, you can pan around the page by dragging it, while the directional pad moves you around all the focusable elements on the page.

What is missing is all the extra stuff that make up a Web browser, such as a navigational toolbar. `WebView` does not provide any of that — if you want those sorts of UI features, you will need to implement those yourself (e.g., use an `EditText` or `AutoCompleteTextView` for a browser address bar).

Supporting JavaScript

Now, you may be tempted to replace the URL in the above source code with something else, such as Google's home page or something else that relies upon JavaScript. You will find that such pages do not work especially well by default. That is because, by default, JavaScript is turned off in `WebView` widgets.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you want to enable JavaScript, call `getSettings().setJavaScriptEnabled(true);` on the `WebView` instance. At this point, any JavaScript referenced by your Web page should work normally.

There are some fancy tricks you can perform with `WebView` and JavaScript, such as having JavaScript call Java code or vice versa. These techniques will be covered in a later chapter.

Alternatives for Loading Content

`loadUrl()` works with:

- `http://` and `https://` URLs
- `file://` URLs pointing to the local filesystem
- `file:///android_asset/` URLs pointing to one of your application's assets, as will be discussed later in this book
- `content://` URLs pointing to a `ContentProvider` that is publishing content available for streaming, as will be discussed much later in this book

Instead of `loadUrl()`, you can also use `loadData()`. Here, you supply the HTML for the `WebView` to display. You might use this to:

1. display a manual that was installed as a file with your application package
2. display snippets of HTML you retrieved as part of other processing, such as the description of an entry in an Atom feed
3. generate a whole user interface using HTML, instead of using the Android widget set

There are two flavors of `loadData()`. The simpler one allows you to provide the content, the MIME type, and the encoding, all as strings. Typically, your MIME type will be `text/html` and your encoding will be `UTF-8` for ordinary HTML.

For example, if you replace the `loadUrl()` invocation in the previous example with the following:

```
browser.loadData("<html><body>Hello, world!</body></html>",
                 "text/html", "UTF-8");
```

You get:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 198: The Browser2 sample application

This is also available as a fully-buildable sample, as `WebKit/Browser2`.

There is also a `loadDataWithBaseURL()` method. This takes, among other parameters, the "base URL" to use when resolving relative URLs in the HTML. Any relative URL (e.g., `<img src="images/foo.png">`) will be interpreted as being relative to the base URL supplied to `loadDataWithBaseURL()`. If you find that you have content that refuses to load properly with `loadData()`, try `loadDataWithBaseURL()` with a `null` base URL, as sometimes that works better, for unknown reasons.

Listening for Events

Particularly if you are going to use the `WebView` as a local user interface (vs. browsing the Web), you will want to be able to get control at key times, particularly when users click on links. You will want to make sure those links are handled properly, either by loading your own content back into the `WebView`, by submitting an `Intent` to Android to open the URL in a full browser, or by some other means. We will discuss using an `Intent` to launch a Web browser in a later chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

One hook into the `WebView` activity is via `setWebViewClient()`, which takes an instance of a `WebViewClient` implementation as a parameter. The supplied callback object will be notified of a wide range of events, ranging from when parts of a page have been retrieved (`onPageStarted()`, etc.) to when you, as the host application, need to handle certain user- or circumstance-initiated events, such as:

1. `onTooManyRedirects()`
2. `onReceivedHttpAuthRequest()`
3. etc.

A common hook will be `shouldOverrideUrlLoading()`, where your callback is passed a URL (plus the `WebView` itself) and you return `true` if you will handle the request or `false` if you want default handling (e.g., actually fetch the Web page referenced by the URL). In the case of a feed reader application, for example, you will probably not have a full browser with navigation built into your reader, so if the user clicks a URL, you probably want to use an `Intent` to ask Android to load that page in a full browser. But, if you have inserted a "fake" URL into the HTML, representing a link to some activity-provided content, you can update the `WebView` yourself.

For example, let's amend the first browser example to be an application that, upon a click, shows the current time.

From [WebKit/Browser3](), here is the revised Java:

```
package com.commonsware.android.webkit;

import android.app.Activity;
import android.os.Bundle;
import android.text.format.DateUtils;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import java.util.Date;

public class BrowserDemo3 extends Activity {
  WebView browser;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);
    browser.setWebViewClient(new Callback());

    loadTime();
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  void loadTime() {
    String page=
        "<html><body><a href='http://webview.used.to.be.less.annoying/clock'>"
            + DateUtils.formatDateTime(this, new Date().getTime(),
                                       DateUtils.FORMAT_SHOW_DATE
                                           | DateUtils.FORMAT_SHOW_TIME)
            + "</a></body></html>";

    browser.loadData(page, "text/html", "UTF-8");
  }

  private class Callback extends WebViewClient {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
      loadTime();

      return(true);
    }
  }
}
```

Here, we load a simple Web page into the browser (`loadTime()`) that consists of the current time, made into a hyperlink to a fake URL. We also attach an instance of a `WebViewClient` subclass, providing our implementation of `shouldOverrideUrlLoading()`. In this case, no matter what the URL, we want to just reload the `WebView` via `loadTime()`.

Running this activity gives us:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 199: The Browser3 Sample Application

Clicking the link will cause us to rebuild the page with the new time.

Note that we are using a `DateUtils` utility class supplied by Android for formatting our date and time. The big advantage of using `DateUtils` is that this class is aware of the user's settings for how they prefer to see the date and time (e.g., 12- versus 24-hour mode).

There is also a `WebChromeClient` that you can register with a `WebView` via a call to `setWebChromeClient()`. This object will be called when various things occur in the `WebView` that might pertain to a browser's "chrome" (i.e., the things outside the HTML rendering area). For example, `onJSAlert()` will be called on your `WebChromeClient` when JavaScript code calls `alert()`.

WebView and Android 4.4

Historically, `WebView` was implemented using a version of WebKit independent from any other browser efforts by Google, such as Chrome.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

As of Android 4.4, `WebView` is now based on Google's open source Chromium project and the Blink rendering engine. However, Blink was forked from WebKit, and so the two started from a common implementation.

At a high level, one WebKit-based HTML rendering engine should work similarly to another WebKit-based HTML rendering engine. However, the expression "the devil is in the details" certainly applies here, and there are a variety of changes in the way `WebView` renders content on Android 4.4 compared to prior versions. Some of these changes will be suppressed if your app has an `android:targetSdkVersion` set to 18 or lower.

It is highly recommended that you test your app thoroughly on both pre–4.4 and 4.4+ environments, and again when you eventually raise your `android:targetSdkVersion` to 19 or higher.

Visit the Trails!

You can learn more about powerful tricks with `WebView`, including integrating the Java and JavaScript environments, in a later chapter.

You can also create apps that run totally in the browser using HTML5, or app frameworks that use `WebView` to render their UI, such as PhoneGap.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Defining and Using Styles

As noted in an earlier chapter, Android offers styles and themes, filling the same sort of role that CSS does in Web development. In that earlier chapter, we covered the basic roles of styles and themes, plus introduced the three classic theme families:

- `Theme`
- `Theme.Holo`
- `Theme.Material`

In this chapter, we will take a slightly "deeper dive" into styles and themes, exploring how you can create your own and apply them to your app's UI.

Styles: DIY DRY

The purpose of styles is to encapsulate a set of attributes that you intend to use repeatedly, conditionally, or otherwise wish to keep separate from your layouts proper. The primary use case is "don't repeat yourself" (DRY) — if you have a bunch of widgets that look the same, use a style to use a single definition for "look the same", rather than copying the look from widget to widget.

And that paragraph will make a bit more sense if we look at an example, specifically the `Styles/NowStyled` sample project. This is a trivial project, with a full-screen button that shows the date and time of when the activity was launched or when the button was pushed. This time, though, we want to change the way the text on the face of the button appears, and we will do so using a style.

The `res/layout/main.xml` file in this project has a `style` attribute on the `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:id="@+id/button"
  android:text=""
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  style="@style/bigred"
/>
```

Note that the `style` attribute is part of stock XML and therefore is not in the `android` namespace, so it does not get the `android:` prefix.

The value, `@style/bigred`, points to a style resource. Style resources are values resources and can be found in the `res/values/` directory in your project, or in other resource sets (e.g., `res/values-v11/` for values resources only to be used on API Level 11 or higher). The convention is for styles resources to be held in a `styles.xml` file, such as the one from the `NowStyled` project:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="bigred">
    <item name="android:textSize">30sp</item>
    <item name="android:textColor">#FFFF0000</item>
  </style>
</resources>
```

The `<style>` element supplies the name of the style, which is what we use when referring to the style from a layout. The `<item>` children of the `<style>` element represent values of attributes to be applied to whatever the style is applied towards — in our example, our `Button` widget. So, our `Button` will have a comparatively large font (`android:textSize` set to `30sp`) and have the text appear in red (`android:textColor` set to `#FFFF0000`).

Just defining the style and applying it to the widget gives us the desired results:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 200: The Styles/NowStyled sample application

Elements of Style

There are four elements to consider when applying a style:

- Where do you put the style attributes to say you want to apply a style?
- What attributes can you define via a style?
- How do you inherit from a previously-defined style (one of your own or one from Android)?
- What values can those attributes have in a style definition?

Where to Apply a Style

The `style` attribute can be applied to a widget, to only affect that widget.

The `style` attribute can be applied to a container, to affect that container. However, doing this does not automatically style its children. For example, suppose `res/layout/main.xml` looked instead like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    style="@style/bigred">
  <Button
    android:id="@+id/button"
    android:text=""
    android:layout_width="match_parent"
    android:layout_height="match_parent"
  />
</LinearLayout>
```

The resulting UI would not have the `Button` text in a big red font, despite the `style` attribute. The style only affects the container, not the contents of the container.

You can also apply a style to an activity or an application as a whole, though then it is referred to as a "theme", which will be covered a bit later in this chapter.

The Available Attributes

When styling a widget or container, you can apply any of that widget's or container's attributes in the style itself. So, if it shows up in the "XML Attributes" or "Inherited XML Attributes" portions of the Android JavaDocs, you can put it in a style.

Note that Android will ignore invalid styles. So, had we applied the `bigred` style to the `LinearLayout` as shown above, everything would run fine, just with no visible results. Despite the fact that `LinearLayout` has no `android:textSize` or `android:textColor` attribute, there is no compile-time failure nor a runtime exception.

Also, layout directives, such as `android:layout_width`, can be put in a style.

Inheriting a Style

You can also indicate that you want to inherit style attributes from another style, by specifying a `parent` attribute on the `<style>` element.

For example, take a look at this style resource:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated" parent="android:Theme.Holo">
    <item name="android:background">?android:attr/
activatedBackgroundIndicator</item>
  </style>
</resources>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

(note: in some renditions of this book, you may see the `<item>` element split over two lines — this is caused by word-wrapping, as this element should be all on one line)

Here, we are indicating that we want to inherit the `Theme.Holo` style from within Android. Hence, in addition to all of our own attribute definitions, we are specifying that we want all of the attribute definitions from `Theme.Holo` as well.

In many cases, this will not be necessary. If you do not specify a parent, your attribute definitions will be blended into whatever default style is being applied to the widget or container.

That `?android:attr` looks a bit bizarre, but we will get into what that syntax means in the next section.

The Possible Values

Typically, the value that you will give those attributes in the style will be some constant, like `30sp` or `#FFFF0000`.

Sometimes, though, you want to perform a bit of indirection — you want to apply some other attribute value from the theme you are inheriting from. In that case, you will wind up using the somewhat cryptic `?android:attr/` syntax, along with a few related magic incantations.

For example, let's look again at this style resource:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated" parent="android:Theme.Holo">
    <item name="android:background">?android:attr/
activatedBackgroundIndicator</item>
  </style>
</resources>
```

Here, we are indicating that the value of `android:background` is not some constant value, or even a reference to a drawable resource (e.g., `@drawable/my_background`). Instead, we are referring to the value of some other attribute — `activatedBackgroundIndicator` — from our inherited theme. Whatever the theme defines as being the `activatedBackgroundIndicator` is what our background should be.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This portion of the Android style system is very under-documented, to the point where Google itself recommends you look at the Android source code listing the various styles to see what is possible.

This is one place where inheriting a style becomes important. In the example shown in this section, we inherited from `Theme.Holo`, because we specifically wanted the `activatedBackgroundIndicator` value from `Theme.Holo`. That value might not exist in other styles, or it might not have the value we want.

Themes: Would a Style By Any Other Name…

Themes are styles, applied to an activity or application, via an `android:theme` attribute on the `<activity>` or `<application>` element. If the theme you are applying is your own, just reference it as `@style/...`, just as you would in a `style` attribute of a widget. If the theme you are applying, though, comes from Android, typically you will use a value with `@android:style/` as the prefix, such as `@android:style/Theme.Holo.Dialog` or `@android:style/Theme.Holo.Light`.

In a theme, your focus is not so much on styling widgets, but styling the activity itself. For example, here is the definition of `@android:style/Theme.Holo.NoActionBar.Fullscreen`:

```
<!-- Variant of the default (dark) theme that has no title bar and
          fills the entire screen -->
<style name="Theme.Holo.NoActionBar.Fullscreen">
  <item name="android:windowFullscreen">true</item>
  <item name="android:windowContentOverlay">@null</item>
</style>
```

It specifies that the activity should take over the entire screen, removing the status bar on phones (`android:windowFullscreen` set to `true`). It also specifies that the "content overlay" — a layout that wraps around your activity's content view — should be set to nothing (`android:windowContentOverlay` set to `@null`), having the effect of removing the title bar.

What Happens If You Have No Theme

Most of the sample apps that we have examined so far have not defined a theme, either at the `<application>` level or the `<activity>` level. What happens here then depends upon the device that your app runs upon:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- On an Android 1.x or 2.x device, you will get `Theme` as your theme
- On an Android 3.x or 4.x device, if your `minSdkVersion` or `targetSdkVersion` is 11 or higher, you will get `Theme.Holo` as your theme; otherwise, you will stick with `Theme` as your theme
- On an Android 5.0+ device, if your `targetSdkVersion` is 14 or higher, you will get `Theme.Material` as your theme; otherwise, your app behaves as in the 3.x/4.x scenario above

As a result, your app is far from "broken", despite the lack of an explicit theme. It does mean, though, that your app will have a different look on those different Android OS levels, a look that will tend to have your app blend in more with other apps on that same device.

However, once you want to start customizing your theme, you will now run into a problem: having different themes for different OS versions. An Android 2.x device knows nothing about `Theme.Material`, for example, so you cannot simply create a custom theme based on `Theme.Material` and expect it to work. As we will see in a later chapter, the solution winds up being *versioned resources*, where you have different theme definitions for different API levels.

Of course, if your `minSdkVersion` is high enough, resource versioning is less of an issue. For example, if your `minSdkVersion` is 21, all devices that your app runs upon should know about `Theme.Material`, just as if your `minSdkVersion` were 11 or higher, all devices that your app would run on would know about `Theme.Holo`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

JARs and Library Projects

Java has as many, if not more, third-party libraries than any other modern programming language. Here, "third-party libraries" refer to the innumerable JARs that you can include in a server or desktop Java application — the things that the Java SDKs themselves do not provide.

In the case of Android, the virtual machine (VM) at its heart is not precisely Java, and what it provides in its SDK is not precisely the same as any traditional Java SDK. That being said, many Java third-party libraries still provide capabilities that Android lacks natively and therefore may be of use to you in your project, for the ones you can get working with Android's flavor of Java. This chapter explains what it will take for you to leverage such libraries and the limitations on Android's support for arbitrary third-party code.

You might think that JARs are the primary model of code reuse within Android. That's not really the case. The primary model of code reuse within Android is the Android library project. Many reusable components and frameworks are distributed as library projects, and we will see several in the course of this book.

The example described in this chapter is the Android Support package, a key piece of reusable code from Google itself, distributed partly as JARs and partly as an Android library project.

But first, let's talk a bit more about Android and VMs.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Dalvik VM, and a Bit of ART

When you are writing Android applications, you are writing Java source code. You might be thinking that your Android device is running Java bytecode, just as your Web browser might when it runs a Java applet.

Alas, you would be mistaken.

Android does not have a Java VM. Android has the Dalvik VM or ART.

The Dalvik VM is a virtual machine, along the lines of the Java VM, the Parrot VM (Perl), Microsoft's CLR, and so forth. Since each VM has its own bytecode, the Dalvik VM bytecode is not the same as the Java VM bytecode (or the Parrot VM bytecode, etc.).

When you build your project, your Java source code is initially compiled using the standard `javac` compiler. Then, however, the Java VM bytecodes created by `javac` are cross-compiled into Dalvik VM bytecodes, and it is *those* bytecodes that are packaged into your APK file and are executed by Android.

Most of the time, you will not notice the difference. Every now and then, though, you will encounter some issues related to Android's use of Dalvik, and the most prominent of these comes when you try repurposing existing Java code.

ART is a new runtime, available for developer testing in Android 4.4. ART still uses Dalvik bytecodes, but uses them as input for an ahead-of-time (AOT) compiler. Rather than relying on a just-in-time (JIT) compiler, as the Dalvik VM does, to translate Dalvik bytecodes into CPU-specific instructions, ART's AOT compiler converts *all* the bytecodes to instructions at installation time.

Getting the Library

You have two easy choices for integrating third-party Java code into your project: use JARs or use an artifact in a repository. The latter approach is for Android Studio users; any IDE can use JARs. The details vary by IDE.

Android Studio

Ideally, the documentation for the third-party library will tell you how to get it as an artifact and add it to your Android Studio project. Specifically, it should tell you a

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

line that you should add to your `dependencies` closure of your app's `build.gradle` file, such as `compile 'com.squareup.retrofit:retrofit:1.6.1'`. We will get into the details of what this line means much later in the book.

The documentation should also indicate what artifact repository this artifact comes from. It may tell you that the artifact comes from "Maven" or "Maven Central", in which case you will need a `mavenCentral()` line in your `repositories` closure:

```
repositories {
    mavenCentral()
}
```

Or, it may tell you something else to use, if the artifact is from another repository, such as:

```
repositories {
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}
```

So, for example, you might wind up with the following in your app's `build.gradle` file:

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'com.squareup.retrofit:retrofit:1.6.1'
}
```

If you have an artifact name (e.g., `com.squareup.retrofit:retrofit:1.6.1`), and you have no indication of where the artifact comes from, try the `mavenCentral()` option.

If all you have is a JAR file, put it in a `libs/` directory in your project's `app/` folder, and then make sure that your `dependencies` closure has the `compile fileTree...` line in it to pull JARs from `libs/`:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

Much more about Android Studio, Gradle, and dependencies can be found later in the book, but this should help you get started.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Eclipse

If you choose to use an existing JAR, place the JAR in the `libs/` directory in your Android project. And that's it, at least for Eclipse. Your JAR will be automatically added to your build path, and your JAR will be automatically bundled into the APK file that is your Android application. Note that other IDEs might require other steps – please consult the documentation for that IDE.

Also note that the R22 version of the ADT plugin for Eclipse may force you to make some adjustments to the "Order & Export" portion of your project's build path. If it seems like your JAR is not being picked up (e.g., you try referring to classes from the JAR and you get runtime `ClassNotFoundExceptions`), then try this recipe:

- Right-click on the project in Eclipse's Package Explorer and choose Build Path > Configure Build Path from the context menu
- Switch to the "Order and Export" tab
- Check the "Android Private Libraries" entry in the list, if it is not already checked

The Outer Limits

Not all available Java code will work well with Android. There are a number of factors to consider, including:

- *Expected Platform APIs*: Does the code assume a newer JVM than the one Android is based on? Or, does the code assume the existence of Java APIs that ship with J2SE but not with Android, such as Swing?
- *Size*: Existing Java code designed for use on desktops or servers need not worry too much about on-disk size, or, to some extent, even in-RAM size. Android, of course, is short on both. Using third-party Java code, particularly when pre-packaged as JARs, may balloon the size of your application.
- *Performance*: Does the Java code effectively assume a much more powerful CPU than what you may find on many Android devices? Just because a desktop can run it without issue does not mean your average mobile phone will handle it well.
- *Interface*: Does the Java code assume a console interface? Or is it a pure API that you can wrap your own interface around?
- *Operating System*: Does the Java code assume the existence of certain console programs? Does the Java code assume it can use a Windows DLL?

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- *Language Version*: Was the JAR compiled with an older version of Java (1.4.2 or older)? Was the JAR compiled with a different compiler than the official one from Sun (e.g., GCJ)? Was the JAR compiled with Java 8 and has Java 8 bytecodes rather than those compatible with Java 6?
- *Dependencies*: Does the Java code depend on other third-party JARs that might have some of these problems as well? Does the Java code depend upon third-party libraries (e.g., the `org.json` JSON library) that are built into Android, but the third party expects a different version of that library?

One trick for addressing some of these concerns is to use open source Java code, and actually work with the code to make it more Android-friendly. For example, if you are only using 10% of the third-party library, maybe it's worthwhile to recompile the subset of the project to be only what you need, or at least removing the unnecessary classes from the JAR. The former approach is safer, in that you get compiler help to make sure you are not discarding some essential piece of code, though it may be more tedious to do.

JAR Dependency Management

One challenge with reusing JARs is that JARs sometimes depend upon other JARs.

If you are using Android Studio, this is handled for you automatically, if you are using artifacts from a repository as the source of the JARs (e.g., pulling from Maven Central).

If you are using Eclipse, you will need to identify and download all relevant dependent JARs, placing them in `libs/` along with the JARs you originally obtained.

OK, So What is a Library Project?

An Android library project is a special type of Android project designed to share code and resources between Android application projects. It is specifically aimed at developers or teams creating multiple applications from the same code base. Library projects can also be used for reusable components, such as distributing custom widgets, activities, or frameworks to third parties.

The biggest difference between an Android library project and a JAR is that an Android library project is designed to distribute *resources* and *manifest entries* as well as Java code. If all you are looking to distribute is Java code, a JAR works just as

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

well as an Android library project. But if you need to distribute layouts, themes, activities, and the like, an Android library project is the better solution.

A later chapter will describe how to create an Android library project.

Using a Library Project

Given that you have a library project — or have identified one you want to use — you can attach it to a regular Android project, so the regular Android project has access to everything in the library. As you might expect, how you accomplish this depends upon your choice of IDE.

Android Studio

Hopefully, the library project that you are wishing to use is being distributed as an AAR — an Android archive file that contains a compiled version of the library's source code, along with the resources and, if supplied, manifest.

More specifically, hopefully the library project that you are wishing to use is being distributed as an AAR as an artifact in a repository. If so, you can integrate it using the same approach as is described earlier in this chapter for JARs — just add the necessary `compile` statement to the `dependencies` closure.

For example, to add the CWAC-ColorMixer library project to your Android Studio app, you would use:

```
repositories {
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}

dependencies {
    compile 'com.commonsware.cwac:colormixer:0.5.+'
}
```

If the Android library project is being distributed in any other way, adding it to Android Studio becomes substantially more complicated. Those scenarios will be examined in greater detail in upcoming chapters on library projects and Gradle dependencies. In that chapter, we will also go into more details about the structure of that `compile` statement and the versioning rules (e.g., `0.5.+`).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Eclipse

To add a reference to an Android library project in Eclipse, you must first have it in your Eclipse workspace, such as via importing the project. Then, go into the project properties window (e.g., right-click on the project and choose Properties). Click on the Android entry in the list on the left, then click the "Add" button in the Library area. This will let you browse to the directory where your library project resides. You can add multiple libraries and control their ordering with the "Up" and "Down" buttons, or remove a library with the "Remove" button.

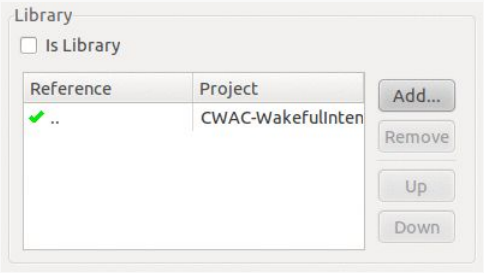


Figure 201: Android Library Project Consumer Properties, Library Section

Library Projects: What You Get

Now, if you build the main project, the Android build tools will:

- Include the `src/` directories of the main project and all of the libraries (`libs/`) in the source being compiled.
- Include all of the resources of the projects, with the caveat that if more than one project defines the same resource (e.g., `res/layout/main.xml`), the highest priority project's resource is included. The main project is top priority, and the priority of the remainder are determined by their order as defined in the build rules (e.g., order in `dependencies` in Android Studio; order in `project.properties` for Eclipse).

This means you can safely reference `R.` constants (e.g., `R.layout.main`) in your library source code, as at compile time it will use the value from the main project's generated `R` class(es).

You may also have some manifest entries automatically injected into your manifest, to add items from a manifest supplied by the library.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Android Support Package

The Android Support package is distributed by Google, containing classes (in JARs and Android library projects) that are not part of the Android SDK, but are available to Android developers.

What's In There?

You can roughly divide the contents of the Android Support package into two major areas:

1. "Backports" of capabilities added to newer versions of Android and the Android SDK, so they can be used on older devices as well. By using the backported classes, you can get the same abilities on a wider range of devices than you could if you only used the classes in the Android SDK.
2. New widgets, containers, or other classes that are not going to be in the Android SDK (for ill-defined reasons) but that Google wishes to make available for Android developers.

More specifically, the most commonly-used pieces of the Android Support package include:

- `support-v4`, which contains backports and miscellaneous UI classes, working back to API Level 4
- `support-v13`, which is identical to `support-v4` but *also* contains a few classes that only work on API Level 13 and higher
- `appcompat-v7`, which is a backport of the action bar, a concept that we will discuss in an upcoming chapter
- `gridlayout-v7`, which is a backport of the `GridLayout` container
- `mediarouter-v7`, which provides a re-implementation of `MediaRouter` and related classes

About the Names

What this book refers to as the "Android Support package" has many names.

It was originally referred to as the Android Compatibility Library, at a time when it only contained backports. Once Google started adding in things that were not strictly related to "compatibility", they started changing the name to try to be more generic. Right now, "Android Support" seems to be fairly consistent, either used

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

standalone or in the form of "Android Support package" or "Android Support Library".

For the purposes of this book, "Android Support package" refers to the entire family of these libraries.

-v4 Versus -v13

Any given project needs *either* `support-v4` *or* `support-v13` (or sometimes neither), not both. If your `minSdkVersion` is 13 or higher, choose the `support-v13` library over the `support-v4` library, as `support-v13` is a clear superset of what is in `support-v4`.

Getting It

You will find the Android Support package in your SDK Manager, in the "Extras" category towards the bottom of the tree:

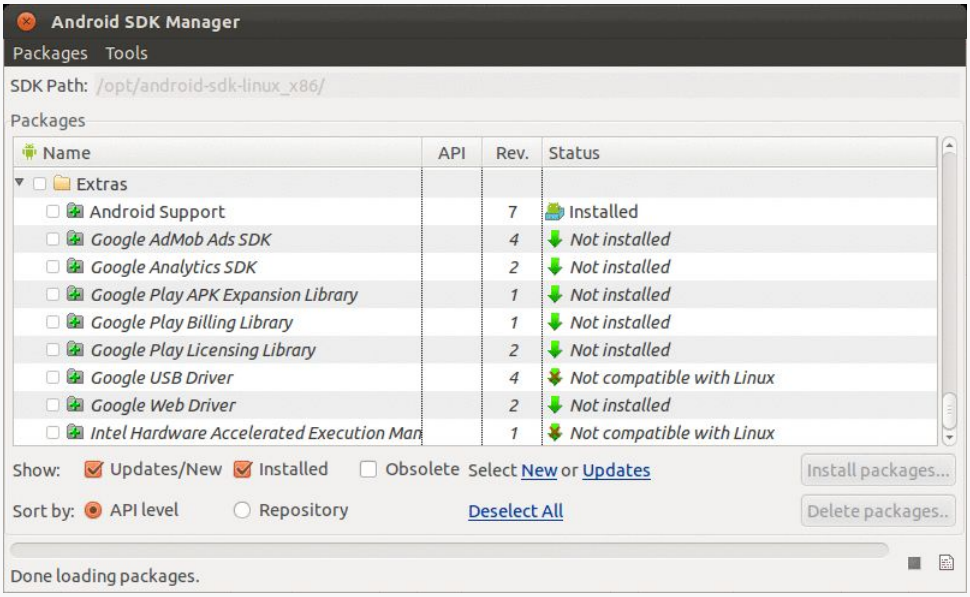


Figure 202: SDK Manager and Android Support Package

To install it, check the checkbox and click the "Install" button, just as you might install an SDK itself.

This will add an `extras/` directory to wherever your SDK installation resides, and the Android Support package will go into subdirectories inside of `extras/`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio users will also want to download the "Android Repository" from the "Extras" area, as this makes the Android Support package's libraries available to you in a local artifact repository, to simplify adding it to your project.

Attaching It To Your Project

As with many things in Android, how you attach Android Support package libraries to your app depends upon your choice of IDE.

Android Studio

You can add references to the Android Support package's libraries — whether those libraries are simple JARs or Android library projects — via a few lines in your `dependencies` closure, referencing the artifacts from the Android Repository.

The easiest way at present to remember what the lines are is to visit the "Gradle, please" Web site, which lists them at the top:

```
// Support Libraries
    compile 'com.android.support:appcompat-v7:21.0.0'
    compile 'com.android.support:cardview-v7:21.0.0'
    compile 'com.android.support:gridlayout-v7:21.0.0'
    compile 'com.android.support:leanback-v17:21.0.0'
    compile 'com.android.support:mediarouter-v7:21.0.0'
    compile 'com.android.support:palette-v7:21.0.0'
    compile 'com.android.support:recyclerview-v7:21.0.0'
    compile 'com.android.support:support-annotations:21.0.0'
    compile 'com.android.support:support-v13:21.0.0'
    compile 'com.android.support:support-v4:21.0.0'
```

Also, while you *could* add all of these to your project, that is not necessary. Only attach dependencies for libraries that you are actually using. Having unused libraries in your project just increases your APK size for no good reason. Hence, most projects will have only a subset of the aforementioned lines.

Eclipse

For `support-v4` or `support-v13`, find the `android-support-v4.jar` or `android-support-v13.jar` file installed in your `extras/` directory tree of your SDK installation and add a copy to your project's `libs/` directory.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The others are distributed as Android library projects. You will need to import the project(s) you want from the Android SDK's `extras/` area, then attach them to your project as is described earlier in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #6 - Adding a Library

We will want to use some third-party libraries in our project, to ease development of the app:

- the Android Support library, specifically its `android-support-v13` JAR
- greenrobot's EventBus, for communication between various pieces of our app
- Google's GSON parser of JSON data
- Square's Retrofit, for retrieving JSON data from Web services

Right now, we will just focus on arranging for our project to be able to use the libraries. Later in the book, we will actually put the libraries to use.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Attaching the Android Support Package

First, we need one of the pieces of the Android Support package, specifically the `support-v13` library, for some classes that we will be using from it later on, such as `ViewPager`.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio

Open the `app/build.gradle` file, off of the project root directory. Find the `dependencies` closure, which should look like this:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

Replace that implementation with:

```
dependencies {
    compile 'com.android.support:support-v13:21.0.3'
}
```

This means that we will no longer be loading JAR files out of a `libs/`, so you can delete that `libs/` directory from the `app/` directory of your project. It also tells Android Studio to incorporate the `support-v13` library from your local Android Repository, which you installed back in Tutorial #1.

Eclipse

In the directory where you installed the Android SDK, find the `extras/android/support/v13` directory, and the `android-support-v13.jar` file inside of there.

Your project should have an empty `libs/` directory. Copy that JAR file into the `libs/` directory. Ideally, you will want to drag-and-drop the file into the `libs/` directory shown in the Eclipse Package Explorer pane, as that not only will copy the file, but it will let Eclipse know about the copy.

Step #2: Downloading the Third-Party JARs

We also need some third-party components for our application eventually, but we can set them up now.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio

The three third-party components we are going to set up now all are available from Maven Central. However, we need to teach Android Studio to be able to load dependencies from Maven Central. To do that, add the following code to your `app/build.gradle` file, above the `dependencies` closure:

```
repositories {
    mavenCentral()
}
```

Then, add three more lines to the `dependencies` closure, identifying the GSON, Retrofit, and EventBus components that we need:

```
dependencies {
    compile 'de.greenrobot:eventbus:2.4.0'
    compile 'com.google.code.gson:gson:2.3.1'
    compile 'com.squareup.retrofit:retrofit:1.8.0'
    compile 'com.android.support:support-v13:21.0.3'
}
```

At this point, your `app/build.gradle` file should look something like:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 19
    buildToolsVersion "21.1.2"

    defaultConfig {
        applicationId "com.commonsware.empublite"
        versionCode 1
        versionName "1.0"
        minSdkVersion 15
        targetSdkVersion 18
    }
}

repositories {
    mavenCentral()
}

dependencies {
    compile 'de.greenrobot:eventbus:2.4.0'
    compile 'com.google.code.gson:gson:2.3.1'
    compile 'com.squareup.retrofit:retrofit:1.8.0'
    compile 'com.android.support:support-v13:21.0.3'
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

At this point, you can save your file.

You may get a yellow banner at the top of the editor, indicating that a "project sync" is requested. If you do, click the "Sync Now" link in that banner to synchronize the `*.iml` files with the changes you made to this `build.gradle` file.

Eclipse

The following URLs point to two JARs that we need (though some do a redirect first):

- greenrobot's EventBus
- Square's Retrofit
- Google's Gson

Copy all three JARs into your `libs/` directory. As before, Eclipse users will want to drag-and-drop the files into the `libs/` directory shown in the Eclipse Package Explorer pane, as that not only will copy the files, but it will let Eclipse know about the copies.

At this point, the `libs/` directory of your project should look like:

Figure 203: Eclipse libs/ Directory Contents

And that's it. Now all four JARs (including the `android-support-v13.jar`) are available to us both at compile-time and at runtime.

In Our Next Episode…

… we will configure the action bar on our tutorial project

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Action Bar

The action bar — that bar that runs across the top of your activity — is the backbone of your UI. Here, you can provide actions for the user to perform related to the current activity (e.g., "edit the contact that you are viewing") or related to the application as a whole (e.g., "here is the documentation"). Sometimes, these actions will appear as toolbar buttons or other widgets in the action bar. Sometimes, these actions will appear in the "overflow", which amounts to a menu.

This chapter introduces the concept of the action bar and how to add actions to it.

Bar Hopping

Android has had many patterns for various "bars" as part of its UI. So, to help explain what an action bar is, it helps if we review the history and role of Android's various bars.

Android 1.x/2.x

In the beginning, there was the status bar and the title bar.

The status bar was a thin strip across the top of the screen, used for things like the clock, signal strength, battery charge, and notification icons (for events like new unread email messages). This bar is technically part of the OS, not your app's UI.

The title bar was a thin gray strip beneath the status bar that, by default, would hold the name of your application, much like the title bar of a browser might show the name of a Web site.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

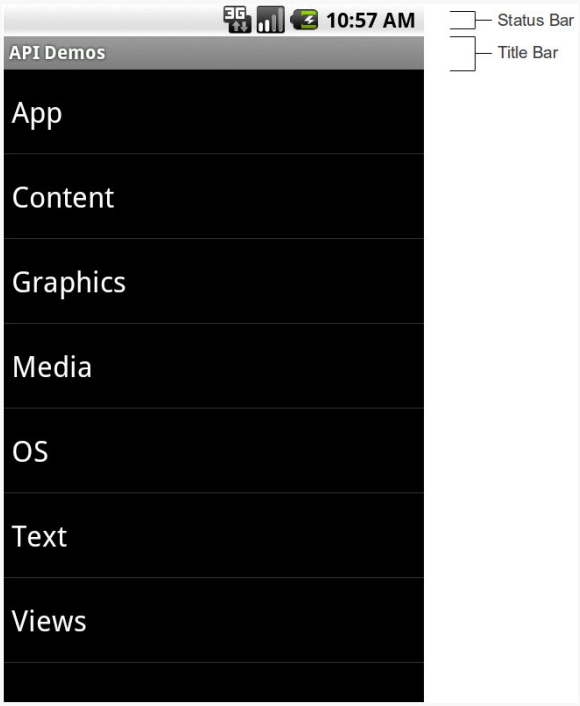


Figure 204: Status Bar and Title Bar

Android 3.0–4.1, Tablets

When official support for tablets arrived with Android 3.0 in February 2011, the story changed.

The status bar was replaced by the system bar, appearing at the bottom of the screen. This had all of the contents of the old status bar, but also had the soft keys for BACK, HOME, etc. Android 1.x and 2.x required that devices have off-screen affordances for those operations; now, device manufacturers could skip those and have the system bar offer them.

The action bar, by default, appears at the top of your activity, replacing the old title bar. You can define what goes in the action bar (icon, title, toolbar buttons, etc.).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

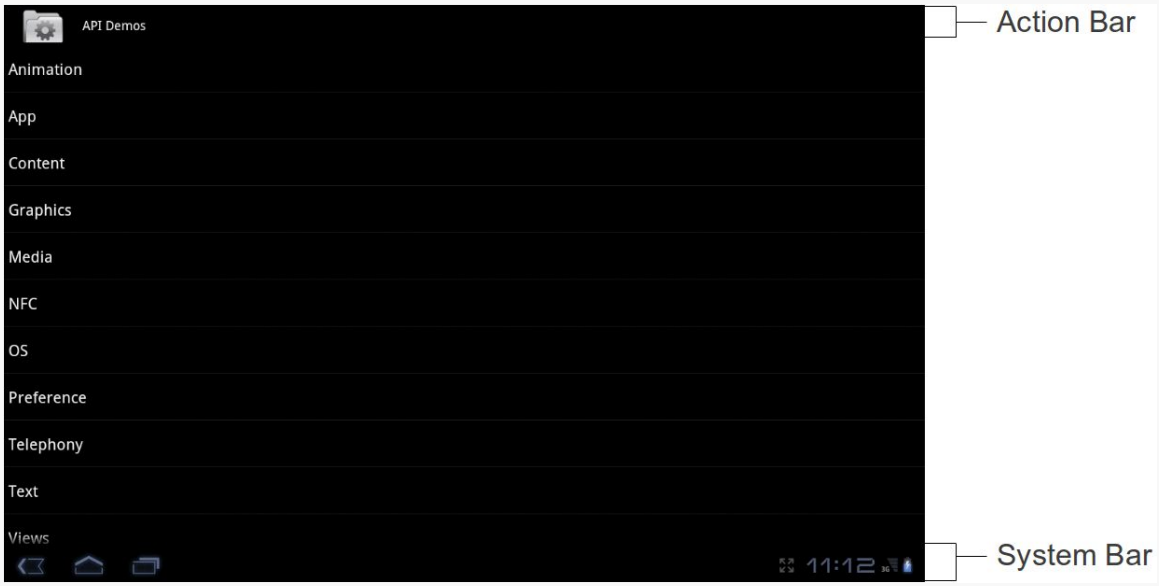


Figure 205: Action Bar and System Bar

The icon on the far left of the action bar also serves as a toolbar button, if you wish. A common pattern for using this is take the user back to the "main" or "home" activity of your application.

Sometimes, the far right side of the action bar will contain a "..." affordance. This is known as the "action overflow" or "overflow menu":

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

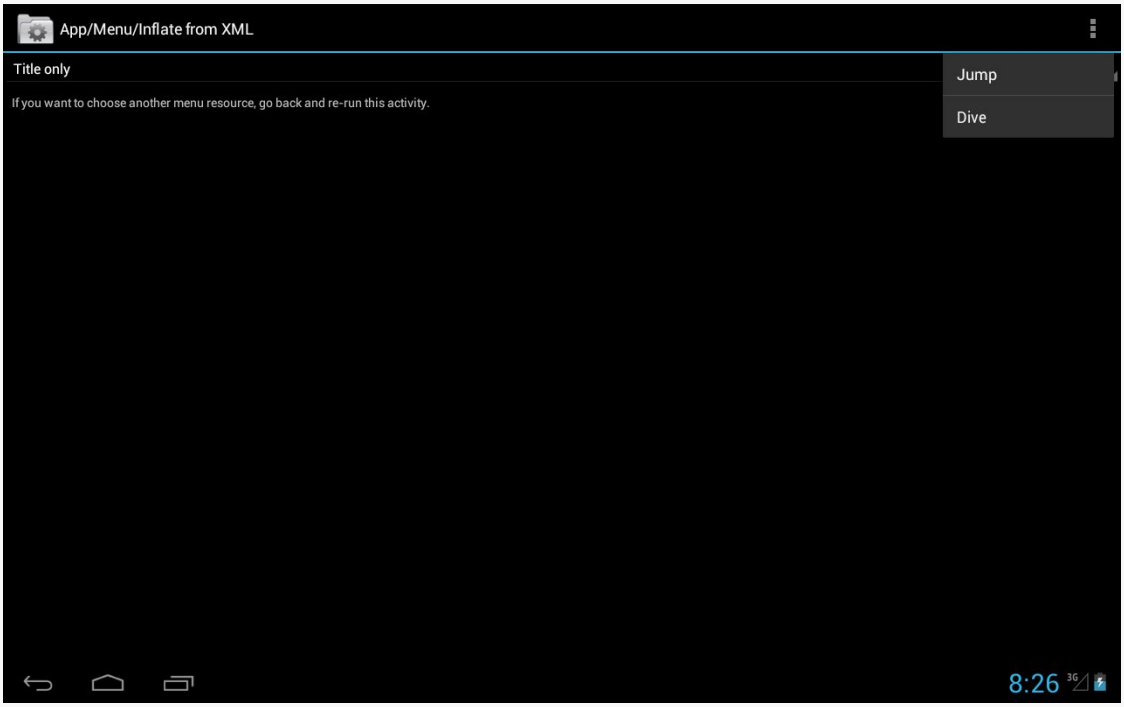


Figure 206: Action Bar with Open Overflow Menu

Tapping it will give the user access to actions that might have been toolbar buttons on a larger screen, but there was insufficient room. Also, low-priority actions may be tucked into the overflow, rather than clutter up the screen with too many toolbar buttons.

Android 4.0–4.4, Phones

Phone-sized devices were not supported by Android 3.x. They jumped from Android 2.3 to 4.0, and along the way adopted some of the Android 3.x UI features:

- Phone apps could have an action bar, like their tablet counterparts
- Device manufacturers could skip the BACK, HOME, etc. buttons and let a partial system bar handle those
- The status bar remained intact from the Android 2.x approach

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

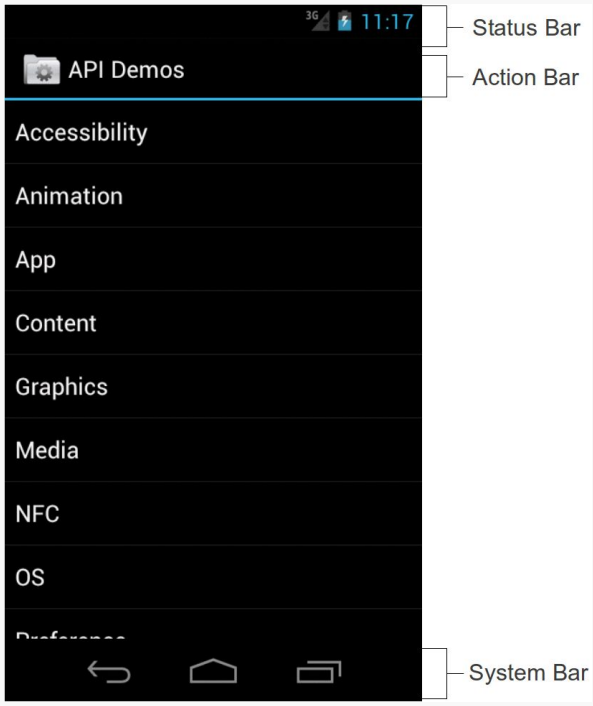


Figure 207: Status Bar, Action Bar, and System Bar

Android 4.2–4.4, Tablets

The Nexus 7, introduced in the summer of 2012, was a 7" tablet that did not follow the tablet UI structure that all other standard Android tablets used. Instead, it looked a bit like a really large phone, having a top status bar along with a bottom system bar solely for the navigation buttons (BACK, HOME, etc.). Apps, as before, could have an action bar as well.

Initially, it was thought that the Nexus 7 was going to be distinctive in that regard. Instead, with Android 4.2, Google switched all tablets to this model, restoring the status bar and relegating the system bar purely for navigation buttons.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

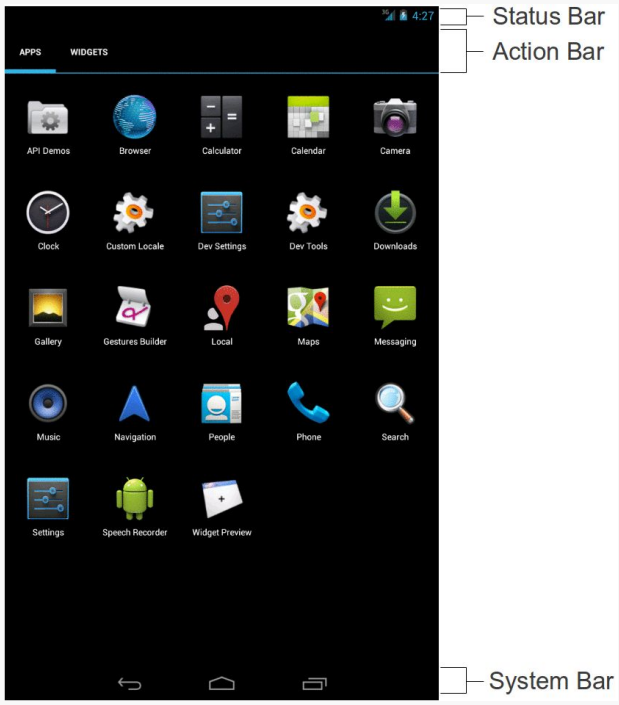


Figure 208: Status Bar, Action Bar, and System Bar, on Nexus 7 Emulator

Android 5.0+

Functionally, the action bar is much the same in Android 5.0 as it was in previous releases. However, aesthetically, it has dropped the icon and made other minor stylistic adjustments.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

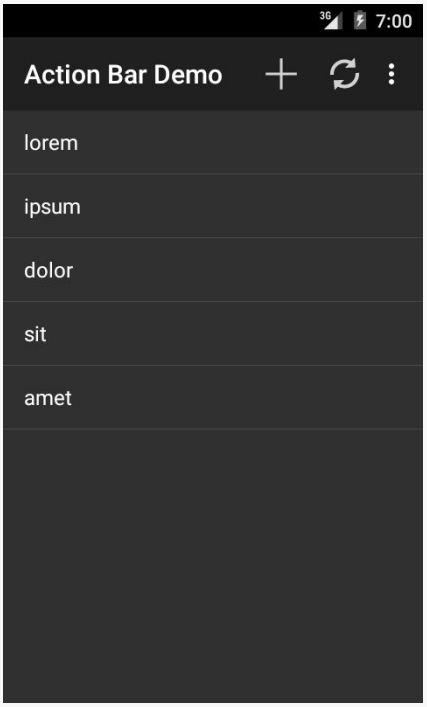


Figure 209: Action Bar on Android 5.0 Emulator

Yet Another History Lesson

Back in the dawn of Android time, referred to by some as "the year 2007", we had options menus. These would rise up from the bottom of the screen based on the user pressing a MENU key:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 210: Legacy Options Menu

This is why you will see references to "options menu" scattered throughout the Android SDK.

The action bar pattern was first espoused by Google at the 2010 Google I|O conference. However, at the time, there was no actual implementation of this, except in scattered apps, and definitely not in the Android SDK.

Android 3.0 — a.k.a., API Level 11 — added the action bar to the SDK, and apps targeting that API level will get an action bar when running on such devices.

Your Action Bar Options

There are several implementations of the action bar floating about. You will probably be using the one that is part of Android itself, starting with API Level 11. However, there are a couple of backports of the action bar if you need them.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Pure Native

As mentioned above, devices running Android 3.0 and higher have support for the action bar as part of their firmware, and that support is exposed through the Android SDK. For example, there is an `ActionBar` class, and you can get an instance of it for your activity's action bar via `getActionBar()`.

However, this only works on devices running Android 3.0 and higher. If you try calling `getActionBar()` on an older device, you will crash with a `VerifyError` runtime exception. `VerifyError` is Android's way of telling you "while you compiled fine, something your compiled code refers to does not exist".

If your `minSdkVersion` is 11 or higher, you will be able to use the native action bar, and that approach will be used in most of this book.

Backports

If your `minSdkVersion` is lower than 11, you have three major choices:

1. Use the "menu" APIs in Android, which will add stuff to the action bar on newer devices, but will result in the classic "options menu" on older devices.
2. Use the `appcompat-v7` backport of the action bar, published by Google in the Android Support package in August 2013.
3. Use ActionBarSherlock, an independent backport of the action bar, published by Jake Wharton in 2011, after Android 3.0 was released.

This chapter assumes that your `minSdkVersion` is set to 11 or higher and you will use the native action bar. Separate chapters in the trails cover the use of `appcompat-v7` and ActionBarSherlock.

Note that, as of October 2014, the `appcompat-v7` library not only backports the action bar, but also attempts to backport part of Google's Material Design styling. Normally, Material Design only comes from Android 5.0 and the use of `Theme.Material`. The `appcompat-v7` chapter will cover the library's effects both to the action bar and to other aspects of your app's UI.

Setting the Target

If you want proper action bar support, you will want to target API Level 14 or higher at runtime. That involves setting the `targetSdkVersion` property in your

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

build.gradle file (for Android Studio users) or setting the android:targetSdkVersion attribute of the <uses-sdk> element of your manifest (for Eclipse users, or Android Studio users maintaining Eclipse compatibility with their projects).

We see this in the manifest of the ActionBar/ActionBarDemoNative sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.inflation"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="10"
    android:targetSdkVersion="19"/>

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name=".ActionBarDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Specifically, we have android:targetSdkVersion set to 19. While 11 or higher will give you an action bar, 14 or higher will solve a particular UI quirk related to menu choices. Some Android 4.0+ devices, but not all, will show *two* ways of getting at overflow menu items if you have your android:targetSdkVersion set to a value between 11 and 13. You will have the "..." item in the action bar itself *and* a second one in the system bar, on devices that have one. Setting android:targetSdkVersion to 14 or higher resolves this.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Doing nothing else but the preceding steps would give us an action bar, but one with no toolbar icons or action overflow menu. While perhaps visually appealing, this is not terribly useful for the user, so we need to do some more work to give the user actions to perform from the action bar.

Note that this manifest has a `minSdkVersion` of 10. This means that the app can run on Android 2.3.3 devices. On those devices, though, the app will not have an action bar, as the action bar did not exist then, and this app is not using a backport like `appcompat-v7`. Instead, the app will have an old-style options menu on API Level 10 devices. There is nothing intrinsically wrong with this, though it does mean that your app will look different on API Level 10 devices.

Defining the Resource

The easiest way to get toolbar icons and action overflow items into the action bar is by way of a menu XML resource. This is called a "menu" resource for historical reasons, as these resources originally were used for things like the options menu.

You can add a `res/menu/` directory to your project and place in there menu XML resources.

Through Eclipse, if you create a new file in there (e.g., `actions.xml`), you will be able to manipulate the menu items using a structured editor, using the "Add" to add a new item and configuring it via the options on the right:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

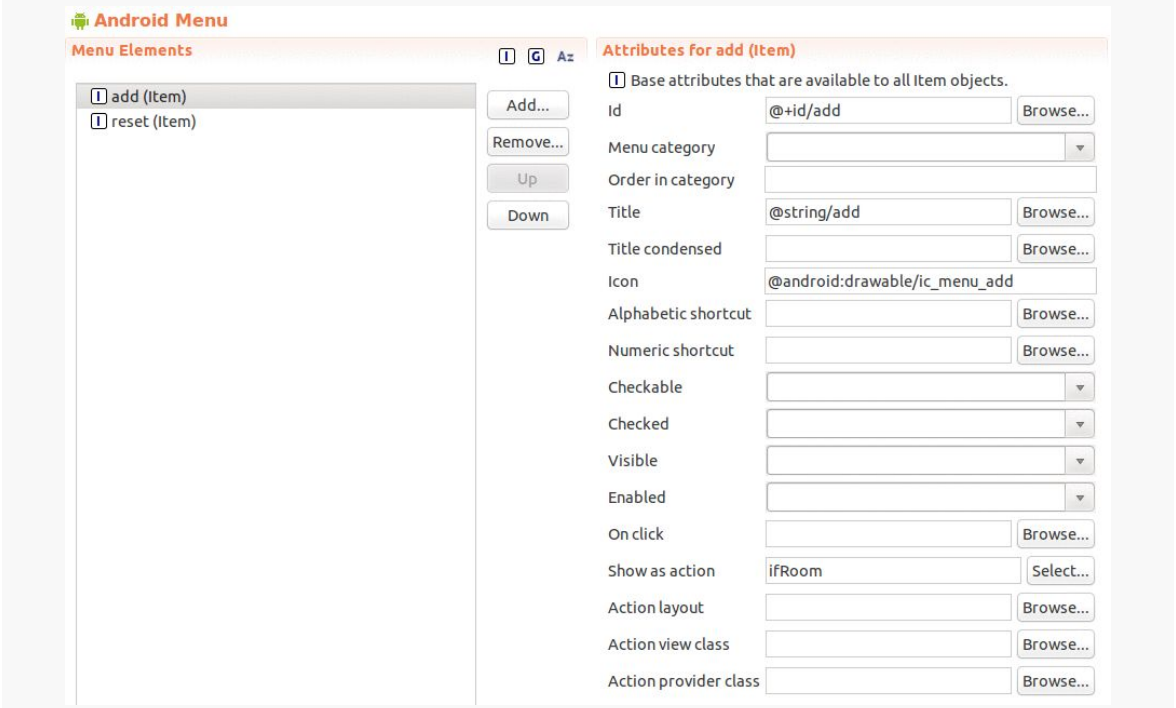


Figure 211: Eclipse Menu Resource Editor

Or, you can work with the raw XML, such as `res/menu/actions.xml` from `ActionBar/ActionBarDemoNative`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/add"
    android:icon="@drawable/ic_action_new"
    android:showAsAction="always"
    android:title="@string/add"/>
  <item
    android:id="@+id/reset"
    android:icon="@drawable/ic_action_refresh"
    android:showAsAction="always|withText"
    android:title="@string/reset"/>
  <item
    android:id="@+id/about"
    android:icon="@drawable/ic_action_about"
    android:showAsAction="never"
    android:title="@string/about">
  </item>

</menu>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you are using Android Studio, you will work with the XML directly.

There are four things you will want to configure on every menu item (`<item>` element in the XML):

1. The ID of the item (via the Id field in Eclipse or the `android:id` attribute in XML). This will create another `R.id` value, associated with this menu item, much like the `R.id` values for our widgets in our layouts. We will use this ID to determine when the user clicks on one of our toolbar buttons or action overflow items.
2. The title of the item (via the Title field in Eclipse or the `android:title` attribute in XML). If this item winds up in the action overflow menu, or optionally as part of its toolbar button, this text will appear. Also, this title will appear as a "tooltip" on the action item in the action bar itself, if the user long-presses on the icon (something few users know to do). Typically, you will use a string resource reference (e.g., `@string/add`), to better support internationalization.
3. The icon for the item (via the Icon field in Eclipse or the `android:icon` attribute in XML). If your item will appear as a toolbar button, this icon is used with that button.
4. Flags indicating how this item should be portrayed in the action bar (via the "Show as action" field in Eclipse or the `android:showAsAction` attribute in XML). You will choose to have it be `always` a toolbar button, only be a toolbar button `ifRoom`, or have it `never` be a toolbar button. You can also elect to append `|withText` to either `always` or `ifRoom`, to indicate that you want the toolbar button to be both the icon and the title, not just the icon. Note that `always` is not *guaranteed* to be a toolbar button — if you ask for 100 `always` items, you will not have room for all of them. However, `always` items get priority for space in the action bar over `ifRoom` items.

Applying the Resource

From your activity, you teach Android about these action bar items by overriding an `onCreateOptionsMenu()` method, such as this one from the `ActionBarDemoActivity` of the `ActionBar/ActionBarDemoNative` sample project:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.actions, menu);

  return(super.onCreateOptionsMenu(menu));
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here, we create a `MenuInflater` and tell it to inflate our menu XML resource (`R.menu.actions`) and pour them into the supplied `Menu` object. We then chain to the superclass, returning its result.

Responding to Events

To find out when the user taps on one of these things, you will need to override `onOptionsItemSelected()`, such as the `ActionBarDemoActivity` implementation shown below:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch(item.getItemId()) {
    case R.id.add:
      addWord();

      return(true);

    case R.id.reset:
      initAdapter();

      return(true);

    case R.id.about:
      Toast.makeText(this, R.string.about_toast, Toast.LENGTH_LONG)
           .show();

      return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

You will be passed a `MenuItem`. You can call `getItemId()` on it and compare that value to the ones from your menu XML resource (`R.id.add` and `R.id.reset`). If you handle the event, return `true`; otherwise, return the value of chaining to the superclass' implementation of the method.

If you wish to respond to taps on your application icon, on the left of the action bar, compare `getItemId()` to `android.R.id.home`, as that will be the `MenuItem` used for that particular toolbar button. Note that if you have your `android:targetSdkVersion` set to `14` or higher, you will also need to call `setHomeButtonEnabled(true)` on the `ActionBar` (obtained via a call to `getActionBar()`) to enable this behavior. Note that this icon may not exist, particularly if you are using `Theme.Material` on Android 5.0+.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Rest of the Sample Activity

So, what is it that we really are doing here in `ActionBarDemoActivity`?

In many respects, this is reminiscent of the `ListActivity` demos from an earlier chapter. We have an array of 25 Latin words, and we want to display these in a list.

However, in this case, we are only showing five words at the outset. An "add" action bar item will add additional words out of the main roster of 25 words, until the `ListView` holds all 25. A "reset" action bar item will return us to the original 5 words.

`ActionBarDemoActivity` is a `ListActivity`. However, rather than set up our `ArrayAdapter` directly in the `onCreate()` method as some of the other samples have done, we delegate that work to an `initAdapter()` method. Moreover, that `initAdapter()` method does its work a bit differently than what those other samples did:

```
  private void initAdapter() {
    words=new ArrayList<String>();

    for (int i=0;i<5;i++) {
      words.add(items[i]);
    }

    adapter=
        new ArrayAdapter<String>(this,
                                 android.R.layout.simple_list_item_1,
                                 words);

    setListAdapter(adapter);
  }
```

Rather than create the `ArrayAdapter` straight out of the static `items` array, we create a fresh `ArrayList` and pour the 5 elements from `items` into it, then create the `ArrayAdapter` on the `ArrayList`. This may seem superfluous, but we will take advantage of this approach with our action bar items.

When the user clicks the "reset" item in the action bar, we call `initAdapter()` again, which gives our `ListActivity` a fresh set of 5 Latin words to display:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
      case R.id.add:
        addWord();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      return(true);

    case R.id.reset:
      initAdapter();

      return(true);

    case R.id.about:
      Toast.makeText(this, R.string.about_toast, Toast.LENGTH_LONG)
           .show();

      return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

When the user clicks the "add" item in the action bar, we call an `addWord()` private method, which adds the next word out of the `items` array and appends it to the `ListView`:

```
private void addWord() {
  if (adapter.getCount()<items.length) {
    adapter.add(items[adapter.getCount()]);
  }
}
```

The net result of all of this is that we have an activity with our customized action bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

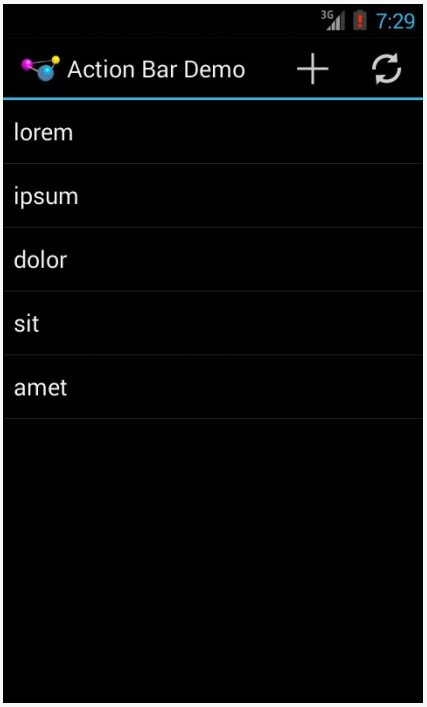


Figure 212: ActionBarDemo, As Initially Launched, on Android 4.3

Among our action bar items is an "about" one that will always be in the overflow menu. This will have three possible visual outcomes.

First, on devices without an off-screen MENU key, the overflow menu is represented by a "..." button, which displays the overflow menu when clicked:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

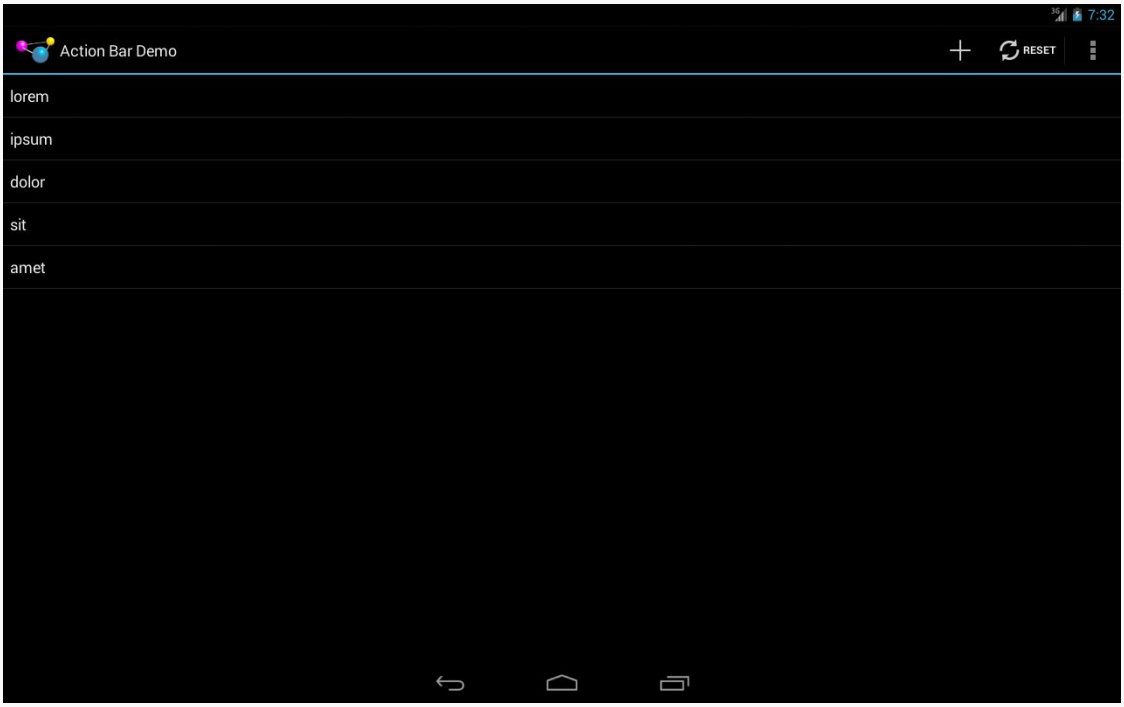


Figure 213: ActionBarDemo, on Android 4.3 Large Screen, with Overflow

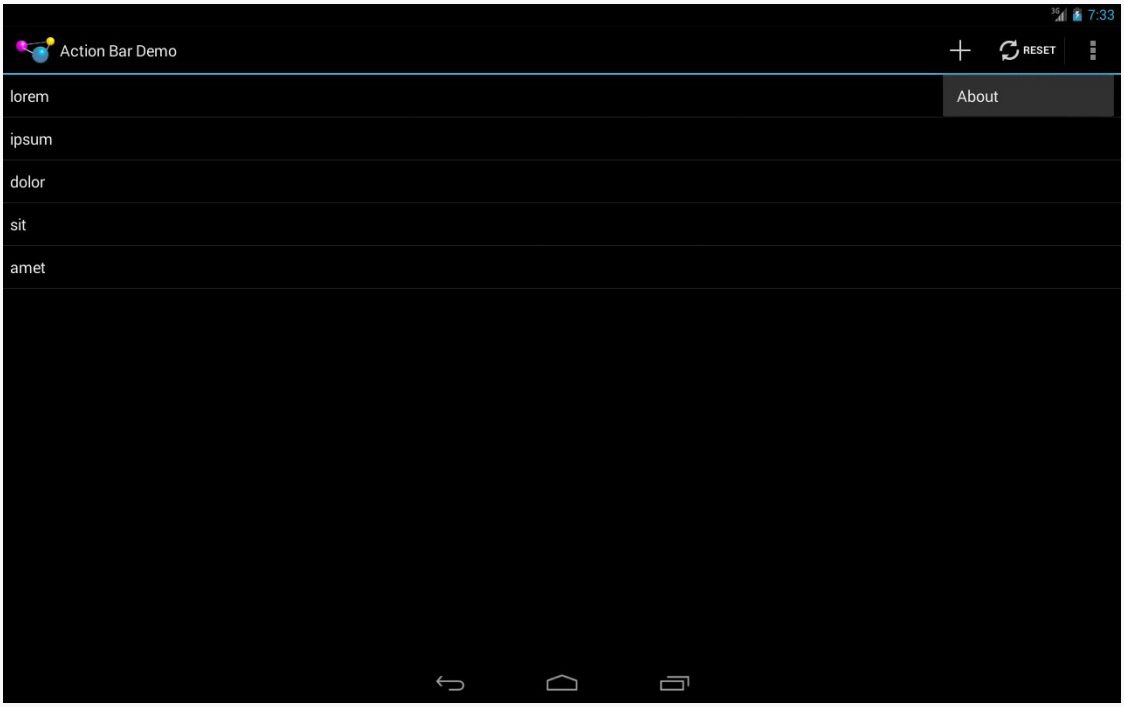


Figure 214: ActionBarDemo, on Android 4.3 Large Screen, with Overflow Open

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

On Android 4.x devices with an off-screen MENU key, pressing the MENU key will cause the overflow menu to rise up from the bottom of the screen:

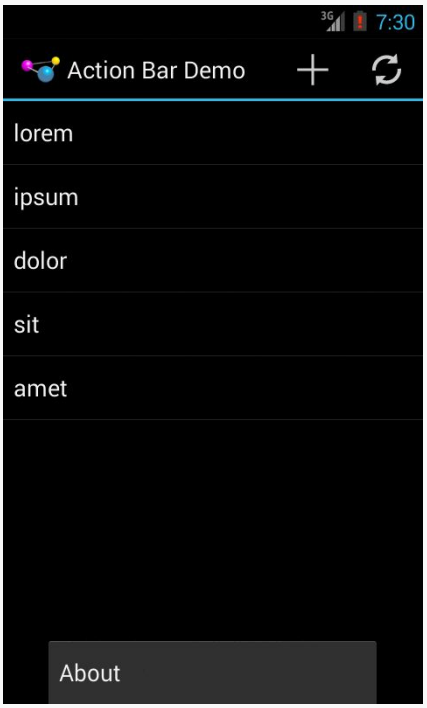


Figure 215: ActionBarDemo, on Android 4.3 Normal Screen, with Overflow

Android 4.4+ devices should always have the "..." button, as is described in the next section.

Android 2.3 devices that run this app will have no action bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 216: ActionBarDemo, on Android 2.3.3 Normal Screen

However, pressing the MENU button will bring up the old-style options menu, where our action items appear:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 217: ActionBarDemo, on Android 2.3.3 Normal Screen, Showing the Options Menu

MENU Key, We Hardly Knew Ye

To expand upon the history lessons from earlier in this chapter, all Android 1.x and 2.x devices had a MENU key, used to bring up the options menu. With Android 3.0 and the advent of the system/navigation bar, device manufacturers no longer needed keys for HOME, BACK, and MENU. And, the action bar incorporated a "..." affordance for accessing the overflow, for items that would have been in the options menu and were not promoted to be toolbar buttons in the action bar itself.

Confusion began when we started having devices that *had* a MENU key *and* Android 3.0+. A few Android 2.x devices were upgraded to Android 4.0, and hundreds of millions of Android devices, from manufacturers like Samsung and HTC, shipped with Android 4.x and a MENU key.

To accommodate this, the device would report whether it had a "permanent menu key", and the action bar would choose whether to show the "..." affordance based upon the existence of this key. Devices with a MENU key would not get the "...", but instead would use the MENU key to display the overflow.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This irritated many developers, for much the same reason as why the MENU key irritated those developers back in Android 1.x/2.x: the existence of a menu was not very discoverable. Many users would eventually realize that tapping the MENU key might uncover useful stuff, but not all users would make this connection. However, now developers could see an obvious alternative, in the form of the "..." affordance, and so they sought ways to trick the action bar into showing the "..." even on devices that had a MENU key.

And that was how the world worked... up until Android 4.4.

An unannounced change in Android 4.4 is that the "..." button should now *always* be shown in the action bar. The MENU key, if it exists, will still work, showing the overflow. Ideally, it shows the overflow as dropping down from the "...", though that is not required. And the Compatibility Definition Document for Android 4.4 more forcefully suggests that the MENU key is obsolete.

None of this should directly affect your code. However:

- When taking screenshots, bear in mind that they will vary between devices that have the "..." button and those that do not
- When writing documentation, or blog posts, or other instructional material, try to phrase references to the overflow that will work for both those users with a "..." button and those that do not

Action Bars, Live in Living Color!

On Android 4.0+, if you are using a `Holo` theme as a base, you may wish to adjust the colors used by your action bar.

On Android 5.0+, if you are using a `Material` theme as a base, you *will* want to adjust the colors used by your action bar. This is Google's vision for how branding should work, in lieu of having your icon be in the action bar.

The following sections outline some ways to affect the colors of your action bar.

Material Tint Effects

Android 5.0 and `Theme.Material` make action bar colors easy to set up, as part of an overall "tinting" approach.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The ActionBar/MaterialColor sample project is a clone of the ActionBarDemoNative sample shown earlier in this chapter, but one where:

- Our minSdkVersion is set to 21, so the app will only run on Android 5.0+
- We set up a custom theme, with specific tinting rules, that affect our action bar colors

Color Resources

The theme will need to refer to colors, and the cleanest way to do that is to set up color resources. Like all of our other resources, we give color resources a name and a color value, usually in #RRGGBB or #AARRGGBB format. Color resources are "value" resources, held by default in res/values/, with the convention of using a colors.xml file for the actual colors.

For example, here is the res/values/colors.xml file from the MaterialColor sample application:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="primary">#3f51b5</color>
  <color name="primary_dark">#1a237e</color>
  <color name="accent">#ffee58</color>
</resources>
```

It defines three colors, primary, primary_dark, and accent, with different colors for each. In Android Studio, editing this file shows a tiny color swatch to help you visualize the colors:

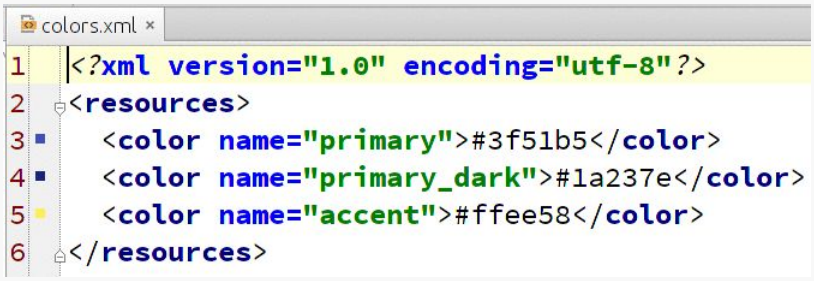


Figure 218: Color Resources in Android Studio

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tinting a Theme

Then, given that we have definitions of our colors, we can apply those colors to a custom theme, found in `res/values/styles.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="AppTheme" parent="android:Theme.Material">
    <item name="android:colorPrimary">@color/primary</item>
    <item name="android:colorPrimaryDark">@color/primary_dark</item>
    <item name="android:colorAccent">@color/accent</item>
  </style>
</resources>
```

Here, our `AppTheme` is inheriting from `Theme.Material` and is overriding three tints: `colorPrimary`, `colorPrimaryDark`, and `colorAccent`, referring to our three color resources in turn.

Note that we could have inherited from `Theme.Material.Light` had we wanted a light "content area" (where our widgets go), or even `Theme.Material.Light.DarkActionBar` for a light content area and a dark action bar (before we start tailoring the action bar colors).

Applying the Theme

The application's manifest declares that we will use `AppTheme` as the default theme for our `<application>`, so all activities will use that theme unless overridden at the activity level:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.abmatcolor"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="21"
    android:targetSdkVersion="21"/>

  <application
    android:allowBackup="false"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name="ActionBarDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Also note that here is where we specify that `minSdkVersion` is 21. An Android Studio project could do that in `build.gradle`, but since Eclipse does not work with Gradle, we have to define it in the manifest.

The Results

Everything else about the app is the same as the `ActionBarDemoNative` sample, including our activity and the `ListView` that we are populating.

However, when we run this edition on an Android 5.0+ device or emulator, our action bar takes on the requested colors, specifically the `colorPrimary` value for the background color of the action bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

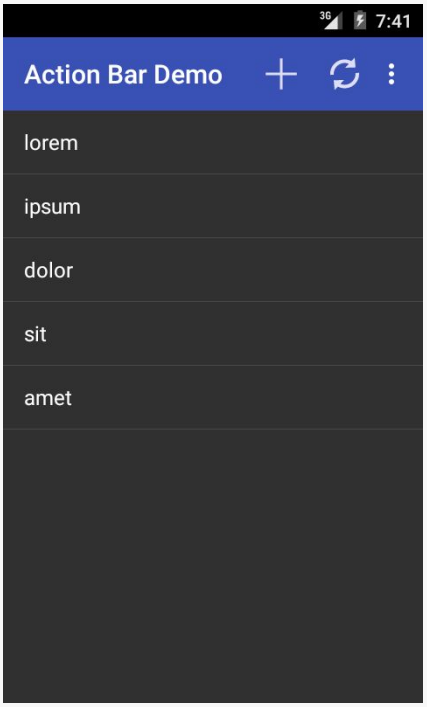


Figure 219: MaterialColor on Android 5.0 Emulator

The custom theme also affects the colors of certain widgets, as will be covered later in the book.

Restoring the Icon (Sort Of)

While the Material Design philosophy skips the application icon that we used to have in the action bar, there is a way to add it back for a `Theme.Material` application, though it requires a little bit of work, as seen in the `ActionBar/MaterialLogo` sample project.

The key thing that you need to do is to call `setDisplayShowHomeEnabled(true)` on your `ActionBar` object, which you get by calling `getActionBar()` in your `Activity`:

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  getActionBar().setDisplayShowHomeEnabled(true);

  initAdapter();
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This will use whatever icon is set for the `android:icon` attribute in your manifest as the "home" icon in your action bar:

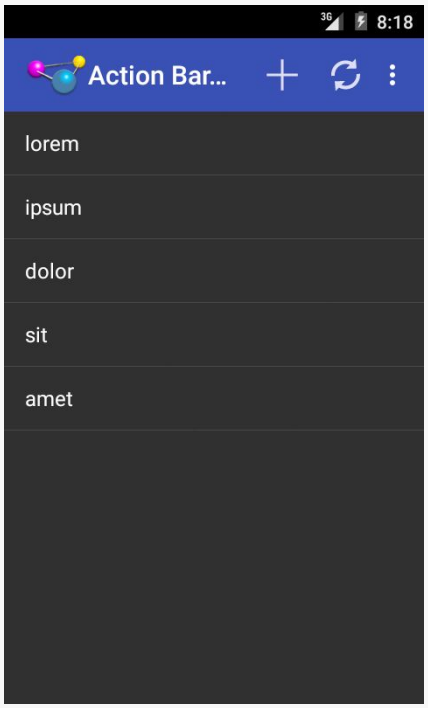


Figure 220: MaterialLogo on Android 5.0 Emulator

If you would rather use a different icon, such as one that is scaled to fit the action bar a bit better, you can call `setIcon()` on your `ActionBar`, supplying the ID of a drawable resource (e.g., `R.drawable.action_bar_icon`) that should be used instead of the drawable specified in the `android:icon` attribute of your `<activity>` or `<application>` in the manifest.

Action Bar Style Generator

For `Theme.Holo` and kin, the tinting rules from `Theme.Material` will not apply. Instead, you will need to do a fair bit of tinkering to get the color scheme set up the way you want.

Or, you can use Jeff Gilfelt's Action Bar Style Generator.

This is a Web site that allows you to design an action bar color scheme, where the site will then generate for you everything that you need to implement that color scheme.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that Mr. Gilfelt has marked this site as deprecated, with an eye towards people using `Theme.Material` or the `appcompat-v7` edition of the action bar. The site works, but in all likelihood it will be discontinued at some future date.

Also note that the site works best with Google's Chrome or Chromium browsers, though in testing, a recent edition of Firefox worked as well. As the site indicates, "your mileage may vary with other browsers".

Designing the Scheme

The site is dominated by form for designing the color scheme and a preview area to show what the design will look like:

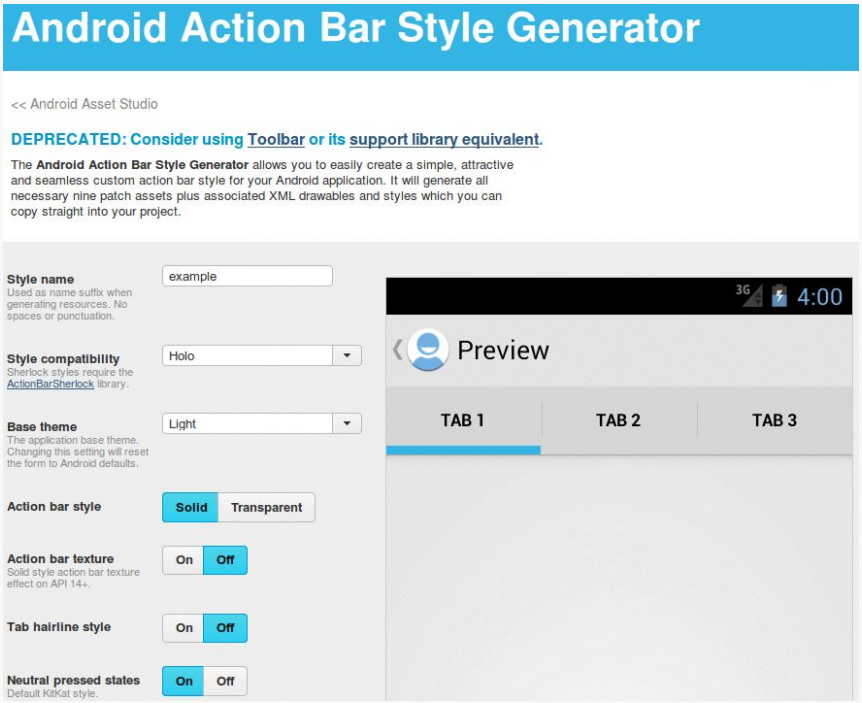


Figure 221: Action Bar Style Generator, As Originally Launched

In the "Style name" field, you can fill in the name you want to give your custom theme. Whatever you fill in will be converted into all lowercase with a leading capital letter, all following `Theme.`. So, for example, filling in `AppTheme` will result in a style resource named `Theme.Apptheme`.

For "Style compatibility", choose "Holo". For "Base theme", choose the base style you want:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Light
- Dark
- Light with dark action bar

The next four options ("Action bar style", "Action bar texture", "Tab hairline style", "Neutral pressed states") are for advanced features and can be left at their defaults.

Scrolling further down the page, you will come to seven color pickers, allowing you to tailor the colors to be used in your action bar implementation. Each picker, when opened, allows you to choose a color based on a fixed palette, then refined using a gradient selector. Or, if you know specific colors (e.g., a graphic designer gave them to you), you can fill the color into the supplied field:

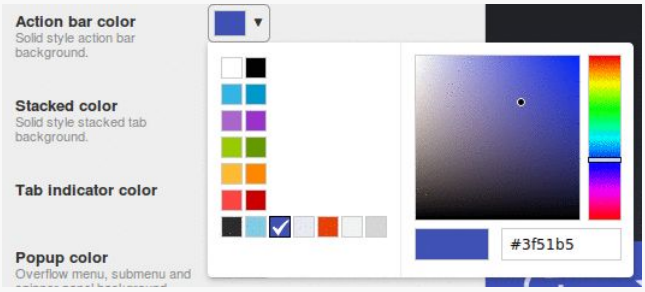


Figure 222: Action Bar Style Generator, Showing "Action bar color" Picker

As you change the colors, you will see what they impact on the preview.

At the bottom of the page is the "Output resources" frame:

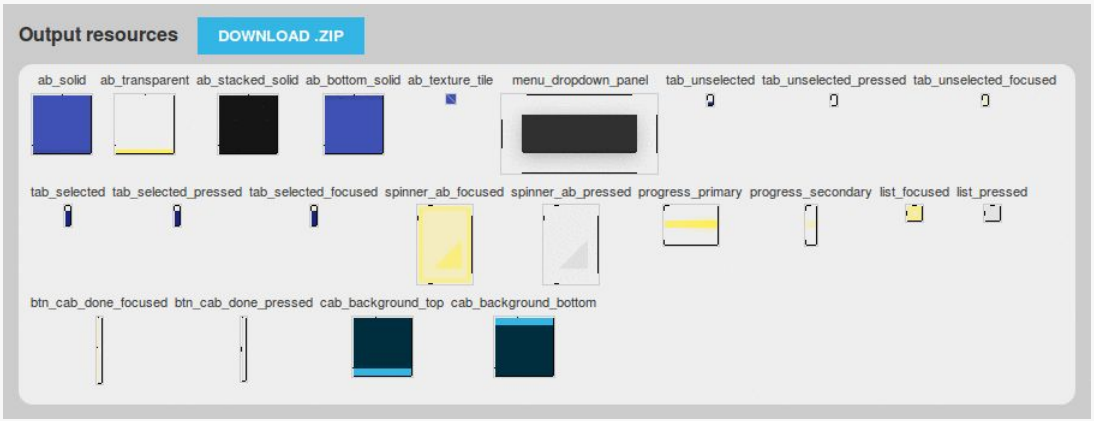


Figure 223: Action Bar Style Generator, Showing "Output resources" Frame

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here, you can click on the "DOWNLOAD .ZIP" button to download a ZIP archive containing your custom theme and all the associated resources required to implement it.

Implementing the Scheme

UnZIP the contents of that ZIP archive into your project's `res/` directory (e.g., in a traditional Android Studio project, unZIP into `src/main/res/` in your `app` module). It will add a bunch of files, notably including a file in `res/values/` whose name is based upon the name you filled into the Web form for the theme name (e.g., `styles_apptheme.xml`).

If you look at that file, you will see that it defines a custom theme for you, named `Theme.` plus whatever you provided to that form (converted into a leading capital letter and the rest lowercase). That file will be rather lengthy, as it designates specific styles to use for various facets of the action bar (e.g., `android:actionBarStyle`).

Here is the theme's primary `<style>` resource element, defining the theme itself:

```
    <style name="Theme.Apptheme" parent="@android:style/Theme.Holo">
        <item name="android:actionBarItemBackground">@drawable/
selectable_background_apptheme</item>
        <item name="android:popupMenuStyle">@style/PopupMenu.Apptheme</item>
        <item name="android:dropDownListViewStyle">@style/
DropDownListView.Apptheme</item>
        <item name="android:actionBarTabStyle">@style/
ActionBarTabStyle.Apptheme</item>
        <item name="android:actionDropDownStyle">@style/
DropDownNav.Apptheme</item>
        <item name="android:actionBarStyle">@style/
ActionBar.Solid.Apptheme</item>
        <item name="android:actionModeBackground">@drawable/
cab_background_top_apptheme</item>
        <item name="android:actionModeSplitBackground">@drawable/
cab_background_bottom_apptheme</item>
        <item name="android:actionModeCloseButtonStyle">@style/
ActionButton.CloseMode.Apptheme</item>


    </style>
```

To use this theme, just add an `android:theme` attribute to your `<application>` (or perhaps individual `<activity>` elements) in your manifest:

```
  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:label="@string/app_name"
    android:theme="@style/Theme.Apptheme">
    <activity
      android:name="ActionBarDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>
```

The resulting app will have a color scheme mirroring what you defined on the form:

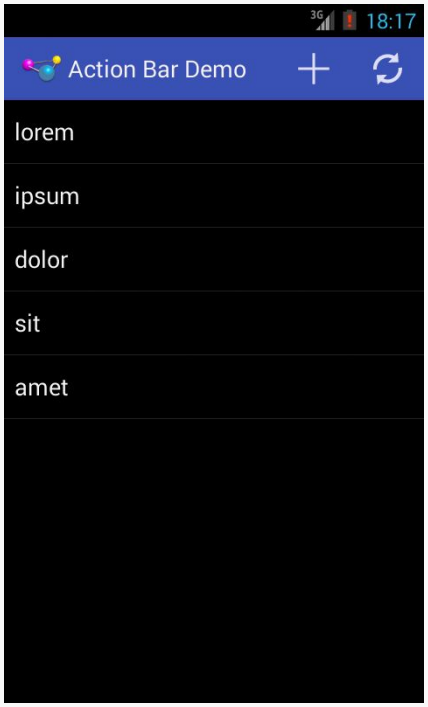


Figure 224: Results of the Action Bar Style Generator

This screenshot, and the code snippets, comes from the ActionBar/HoloColor sample project, which is the same as the base action bar sample app from this chapter with the custom theme applied.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Visit the Trails!

In addition to this chapter, you can learn more about advanced action bar techniques and learn about action modes, which temporarily replace the action bar with new items for use with contextual operations.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #7 - Setting Up the Action Bar

Next up is to configure the action bar to our EmPubLite application.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Starting in this tutorial, we will now begin editing Java source files. Eclipse users may wish to try to remember two useful shortcut key combinations (Windows/Linux syntax shown):

- `<Ctrl>-<Shift>-<O>` will organize your Java `import` statements, including finding imports for any classes or interfaces you have referenced in your code but have not yet imported. In Android Studio, this is `<Ctrl>-<Alt>-<O>`.
- `<Ctrl>-<Shift>-<F>` will reformat the Java or XML in the current editing window, in accordance with either the default styles in Eclipse or whatever you have modified them to via the Preferences window. In Android Studio, this is `<Ctrl>-<Alt>-<L>`.

Step #1: Acquiring Some Icons

As you noticed when running the previous versions of our app, our current action bar is very boring.

Very, very boring.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To make it more useful and worthy of its screen space, we need to start adding some action items. And to do that, we will need some icons.

There are many sources of icons for use in Android apps, but few of them are part of the IDE. In particular, the IDE does not have many icons that are well-suited for use with action bar items. However, Google does publish the "Action Bar Icon Pack", which contains a number of icons already set up for use in the action bar, with the appropriate styling and implementations in various screen densities.

Visit the Design Downloads page and download the "Action Bar Icon Pack". You will find it convenient if you unZIP the archive somewhere on your development machine, though if you prefer to leave it ZIPped that is your choice.

Then, copy the `13_extra_actions_about` icons from the `Action Bar Icons/holo_dark/` directory from the "Action Bar Icon Pack" into your project. In the `13_extra_actions_about` directory, you will see versions of this icon in various densities. Copy those icon files into their corresponding directories in your project. You can do this by dragging the file from your development desktop OS into the project explorer window and dropping it into the proper directory.

Finally, repeat the above process for `13_extra_actions_help`.

Be sure to hold onto the "Action Bar Icon Pack", though, as we will be getting more icons from it in later tutorials.

Step #2: Defining Some Options

Next, we will add a couple of low-priority action items, for a help screen and an "about" screen.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Right click over the `res/` directory in your project, and choose New > "Android resource directory" from the context menu. This will bring up a dialog to let you create a new resource directory:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

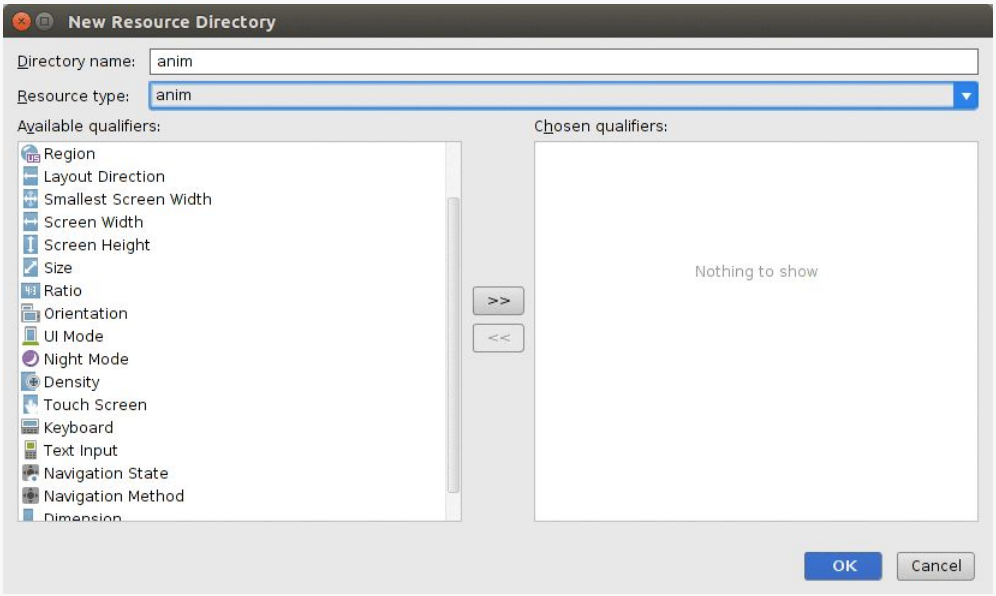


Figure 225: Android Studio New Resource Directory

Change the "Resource type" drop-down to be "menu", then click OK to create the directory.

Then, right-click over your new `res/menu/` directory and choose New > "Menu resource file" from the context menu. Fill in `options.xml` in the "New Menu Resource File" dialog:

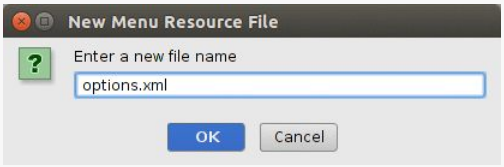


Figure 226: Android Studio New Menu Resource Dialog

Then click OK to create the file. It will open up into an XML editor, into which you can paste the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/help"
        android:icon="@drawable/ic_action_help"
        android:title="@string/help">
    </item>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    <item
        android:id="@+id/about"
        android:icon="@drawable/ic_action_about"
        android:title="@string/about">
    </item>

</menu>
```

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Also, you will need to add string resources for `help` and `about`, by adding appropriate `<string>` elements to your existing `res/values/strings.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">EmPub Lite</string>
    <string name="help">Help</string>
    <string name="about">About</string>
</resources>
```

Eclipse

Open the `res/menu/` folder in your project. Right-click over the `menu` folder and choose New > Other from the context menu. Choose "Android XML File" in the list of possible new-file templates and click Next. In the File field in the next page of the wizard, fill in `options` and click Finish.

This should open up the menu in the structured editor:

Figure 227: Eclipse Menu Editor

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, the structured editor has multiple bugs. Hence, click the "options.xml" sub-tab to switch to the raw XML editor. Then, paste in the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/help"
        android:icon="@drawable/ic_action_help"
        android:title="@string/help">
    </item>
    <item
        android:id="@+id/about"
        android:icon="@drawable/ic_action_about"
        android:title="@string/about">
    </item>

</menu>
```

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Also, you will need to add string resources for `help` and `about`, by adding appropriate `<string>` elements to your existing `res/values/strings.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">EmPub Lite</string>
    <string name="help">Help</string>
    <string name="about">About</string>
</resources>
```

Step #3: Loading and Responding to Our Options

Simply defining `res/menu/options.xml` is insufficient. We need to actually tell Android to use what we defined in that file, and we need to add code to respond to when the user taps on our items.

To do that, you will need to add an `onCreateOptionsMenu()` method and an `onOptionsItemSelected()` method to `EmPubLiteActivity`, as follows:

```
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.options, menu);
        return(super.onCreateOptionsMenu(menu));
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case android.R.id.home:
                return(true);
            case R.id.about:
                return(true);
            case R.id.help:
                return(true);
        }
        return(super.onOptionsItemSelected(item));
    }
}
```

NOTE: Copying and pasting Java code may or may not work, depending on what you are using to read the book. For the PDF, some PDF viewers (e.g., Adobe Reader) should copy the code fairly well; others may do a much worse job.

In `onCreateOptionsMenu()`, we are inflating `res/menu/options.xml` and pouring its contents into the supplied `Menu` object, which will be used by Android to populate our action bar.

In `onOptionsItemSelected()`, we examine the supplied `MenuItem` and route to different branches of a `switch` statement based upon the item's ID. In addition to `R.id.about` and `R.id.help` — for the two items we defined in `res/menu/options.xml`, we also watch for `android.R.id.home`, which eventually will be triggered by a tap on our icon, on the left side of the action bar.

To get this to compile, you will need to add some imports as well:

```
import android.view.Menu;
import android.view.MenuItem;
```

Android Studio users can press `<Alt>-<Enter>` with the cursor in a class reference that is missing its import to add that import. Eclipse users can resolve all missing imports at one time using `<Ctrl>-<Shift>-<O>`.

If you run this in a device or emulator, you may see no initial difference. That would be for devices or emulators that have a MENU button and are running Android 4.3 or below. To display our options, you would need to press MENU:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

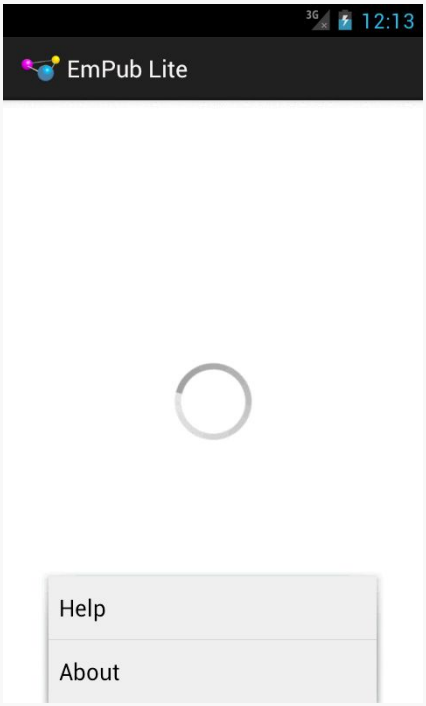


Figure 228: EmPubLite, With Options Via the MENU Button

Note that on an emulator, the MENU button is mapped to the `PgUp` key of your keyboard.

In other cases, the action bar will have a "..." icon on the action bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 229: EmPubLite, Showing the ... Overflow Button

Pressing that brings up a menu showing our items:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

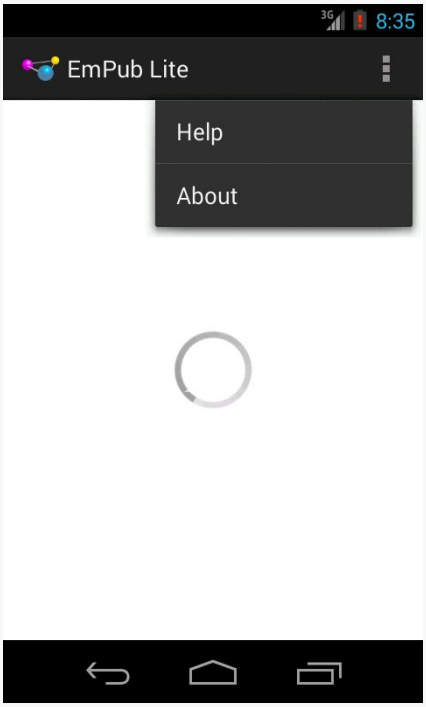


Figure 230: EmPubLite, Showing the Overflow Options

In Our Next Episode…

… we will define our first new activity on the tutorial project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android's Process Model

So far, we have been treating our activity like it is our entire application. Soon, we will start to get into more complex scenarios, involving multiple activities and other types of components, like services and content providers.

But, before we get into a lot of that, it is useful to understand how all of this ties into the actual OS itself. Android is based on Linux, and Linux applications run in OS processes. Understanding a bit about how Android and Linux processes inter-relate will be useful in understanding how our mixed bag of components work within these processes.

When Processes Are Created

A user installs your app, goes to their home screen's launcher, and taps on an icon representing your activity. Your activity dutifully appears on the screen.

Behind the scenes, what happened is that Android forked a copy of a process known as the `zygote`. As a result of the way your process is forked from the `zygote`, your process contains:

- A copy of the VM (Dalvik or ART), shared among all such processes via Linux copy-on-write memory sharing
- A copy of the Android framework classes, like `Activity` and `Button`, also shared via copy-on-write memory
- A copy of your own classes, loaded out of your APK
- Any objects created by you or the framework classes, such as the instance of your `Activity` subclass

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

BACK, HOME, and Your Process

Suppose that you have an app with just one activity. From the home screen's launcher, the user taps on the icon associated with your app's activity. Then, with your activity in the foreground, the user presses BACK.

At this point, the user is telling the OS that she is done with your activity. Control will return to whatever preceded that activity — in this case, the home screen's launcher.

You might think that this would cause your process to be terminated. After all, that is how most desktop operating systems work. Once the user closes the last window of the application, the process hosting that application is terminated.

However, that is not how Android works. Android will *keep* your process around, for a little while at least. This is done for speed and power: if the user happens to want to return to your app sooner rather than later, it is more efficient to simply bring up another copy of your activity again in the existing process than it is to go set up a completely new copy of the process. This does not mean that your process will live forever; we will discuss when your process will go away later in this chapter.

Now, instead of the user pressing BACK, let's say that the user pressed HOME instead. Visually, there is little difference: the home screen re-appears. Depending on the home screen implementation there may be a visible difference, as BACK might return to a launcher whereas HOME might return to something else on the home screen. However, in general, they feel like very similar operations.

The difference is what happens to your activity.

When the user presses BACK, your foreground activity is *destroyed*. We will get into more of what that means in the next chapter. However, the key feature is that the activity itself — the instance of your subclass of `Activity` – will never be used again, and hopefully is garbage collected.

When the user presses HOME, your foreground activity is *not* destroyed... at least, not immediately. It remains in memory. If the user launches your app again from the home screen launcher, and if your process is still around, Android will simply bring your existing activity instance back to the foreground, rather than having to create a brand-new one (as is the case if the user pressed BACK and destroyed your activity).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What HOME literally is doing is bringing the home screen activity back to the foreground, not otherwise directly affecting your process much.

Termination

Processes cannot live forever. They take up a chunk of RAM, for your classes and objects, and these mobile devices only have so much RAM to work with. Eventually, therefore, Android has to get rid of your process, to free up memory for other applications.

How long your process will stick around depends on a variety of factors, including:

- What else the device is doing, either in the foreground (user using apps) or in the background (e.g., automated checks for new email)
- How much memory the device has
- What is still running inside your process

Going back to the scenario from above, we have an application with a single activity launched from the home screen, where the user can return to the home screen either by pressing BACK or by pressing HOME. You might think that this has no difference at all on when the process would be terminated, but that would be incorrect. Pressing HOME would keep the process around perhaps a bit longer than would pressing BACK.

Why?

When the user presses BACK, your one and only activity is destroyed. When the user presses HOME, your activity is not destroyed. Android will tend to keep processes around longer if they have active (i.e., not destroyed) components in them.

The key word there is "tend". Android's algorithms for determining when to get rid of what processes are baked into the OS and are, at best, lightly documented. There is evidence to suggest that other criteria, such as process age, are also taken into account, and so there may be times when a process that has an activity running (but not in the foreground) might be terminated where a process with no running activity might not. However, in general, processes with active (not destroyed) components will stick around a bit longer than processes without such components.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Foreground Means "I Love You"

Just because Android terminates processes to free up memory does not mean that it will terminate just any process to free up memory. A foreground process – the most common of which is a process that has an activity in the foreground – is the least likely of all to be terminated. In fact, you can pretty much assume that if Android has to kill off the foreground process, that the phone is very sick and will crash in a matter of moments.

(and, fortunately, that does not happen very often)

So, if you are in the foreground, you are safe. It is only when you are not in the foreground that you are at risk of having the process be terminated.

You and Your Heap

Processes take up RAM. A significant chunk of that RAM represents the objects you create (a.k.a., "the heap").

Those of you with significant Java backgrounds know that the Java VM loves RAM ("can't get enough of it!"). Java VMs routinely grab 64MB or 128MB of heap space upon creating the process and will grow as big as you wish to let them (e.g., `-Xmx` switch to the `java` command).

Android heap sizes are not that big, because Android is designed to run on mobile devices with constrained amounts of RAM.

Your heap limit may be as low as 16MB, though values in the 32–48MB range are more typical with current-generation devices. How much the heap limit will be depends a bit on what version of Android is on the device. It depends quite a lot, though, on the screen size, as bigger screens will tend to want to display bigger bitmap images, and bitmap images can consume quite a bit of RAM.

The key is that the heap is small, and (generally speaking) you cannot adjust it yourself. It is what it is. Small applications will rarely run into a problem with heap space, but larger applications might. We will discuss tools and techniques for measuring and coping with memory problems later in this book.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Activities and Their Lifecycles

An Android application will have multiple discrete UI facets. For example, a calendar application needs to allow the user to view the calendar, view details of a single event, edit an event (including adding a new one), and so forth. And on smaller-screen devices, like most phones, you may not have room to squeeze all of this on the screen at once.

To handle this, you can have multiple activities. Your calendar application may have one activity to display the calendar, another to add or edit an event, one to provide settings for how the calendar should work, another for your online help, etc. Some of these activities might be private to your app, while others might be able to be launched by third parties, such as your "launcher" activity being available to home screens.

All of this implies that one of your activities has the means to start up another activity. For example, if somebody clicks on an event from the view-calendar activity, you might want to show the view-event activity for that event. This means that, somehow, you need to be able to cause the view-event activity to launch and show a specific event (the one the user clicked upon).

This can be further broken down into two scenarios:

- You know what activity you want to launch, probably because it is another activity in your own application
- You have a reference to... something (e.g., a Web page), and you want your users to be able to do... something with it (e.g., view it), but you do not know up front what the options are

This chapter will cover both of those scenarios.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In addition, frequently it will be important for you to understand when activities are coming and going from the foreground, so you can automatically save or refresh data, etc. This is the so-called "activity lifecycle", and we will examine it in detail as well in this chapter.

Creating Your Second (and Third and…) Activity

Unfortunately, activities do not create themselves. On the positive side, this does help keep Android developers gainfully employed.

Hence, given a project with one activity, if you want a second activity, you will need to add it yourself. The same holds true for the third activity, the fourth activity, and so on.

The sample we will examine in this section is `Activities/Explicit`. Our first activity, `ExplicitIntentsDemoActivity`, started off as just the default activity code generated by the build tools. Now, though, its layout contains a `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <Button
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:textSize="20sp"
    android:text="@string/hello"
    android:onClick="showOther"/>

</LinearLayout>
```

That `Button` is tied to a `showOther()` method in our activity implementation, which we will examine shortly.

Defining the Class and Resources

To create your second (or third or whatever) activity, you first need to create the Java class. You need to create a new Java source file, containing a public Java class that extends `Activity` directly or indirectly. You have three basic ways of doing this:

- Just create the class and resources yourself
- Use the Android Studio new-activity wizard

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Use the Eclipse new-activity wizard

Android Studio New-Activity Wizard

Right-click on your `app/src/main/` sourceset directory in the project explorer, and go into the New > Activity portion of the context menu. This will give you a submenu of available activity templates — mostly the same roster of templates that we saw back when we created the project in the first place.

If you choose one of those templates, you will be presented with a one-page wizard in which to provide the details for this activity:

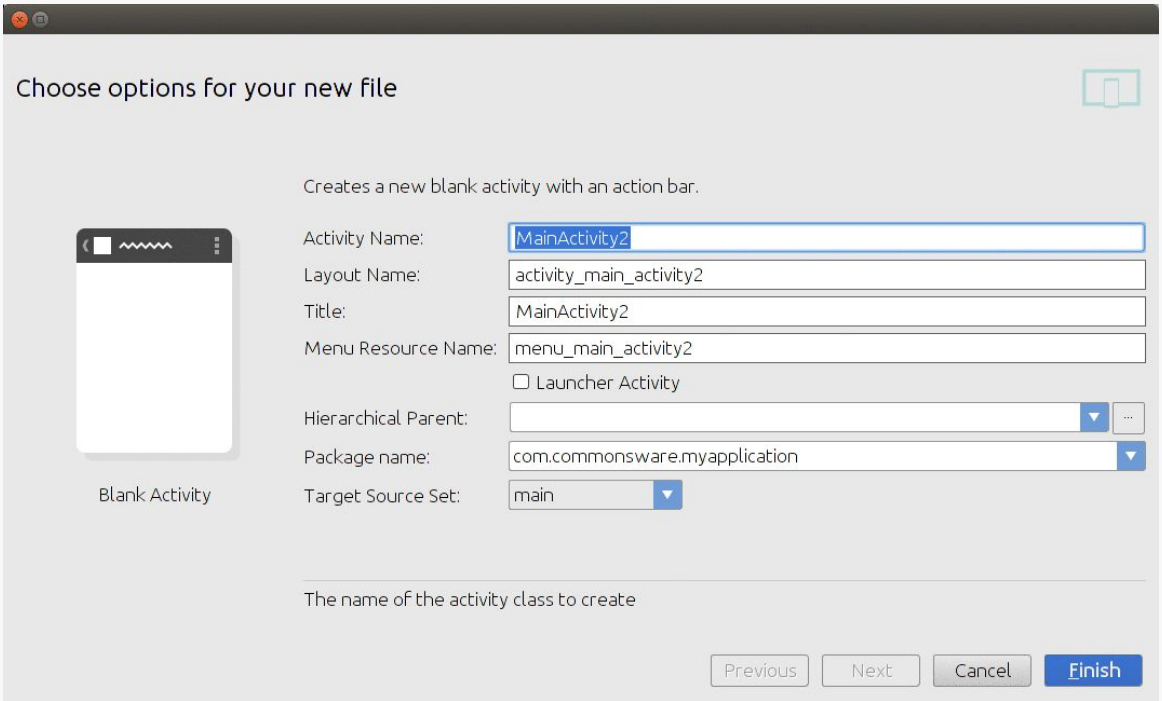


Figure 231: Android Studio New-Activity Wizard

The first few fields will be based upon the template you chose (e.g., activity name, layout XML resource name) and will mirror those we saw back in the new-project wizard. Others, though, are new to this wizard:

- the "Launcher Activity" checkbox which, if checked, will add the appropriate bits to the manifest to tell Android that this activity should get an icon in the home screen's launcher

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- the “Hierarchical Parent” drop-down, which presently is beyond the scope of this book
- the “Package name” drop-down, where you can choose the Java package name into which this activity’s class will be added
- the “Target Source Set” drop-down, where you can indicate the Gradle sourceset into which the generated code should go (for now, choose `main`)

Clicking “Finish” will then create the activity’s Java class, related resources, and manifest entry.

Eclipse New-Activity Wizard

Right-clicking over a project in the Package Explorer, and choosing New > Other from the context menu, brings up a list of things that you can add to your project. One of those, in the Android group, is “Android Activity”:

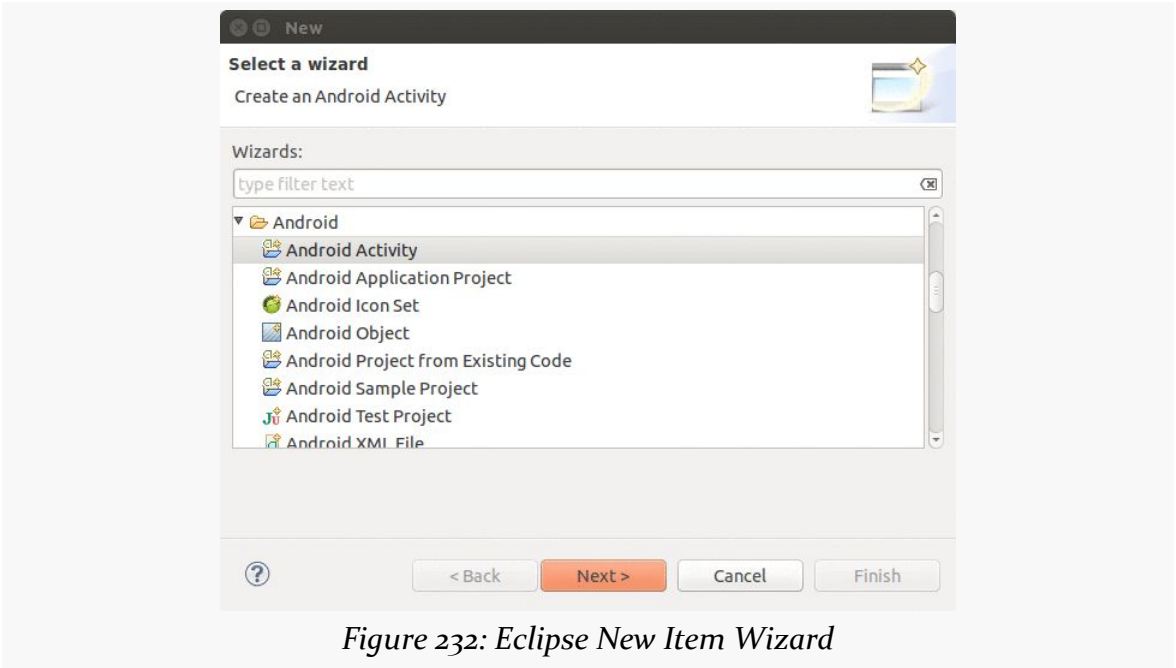


Figure 232: Eclipse New Item Wizard

Choosing “Android Activity” and clicking Next will give you a list of available activity templates:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

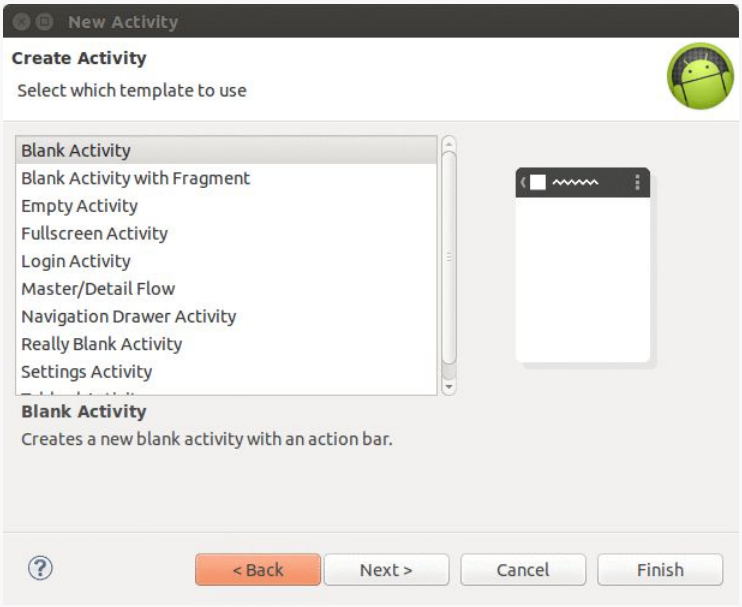


Figure 233: Eclipse New Activity Wizard, First Page

Choosing a template and clicking Next will bring up another wizard page, where you will need to provide additional information about the new activity:

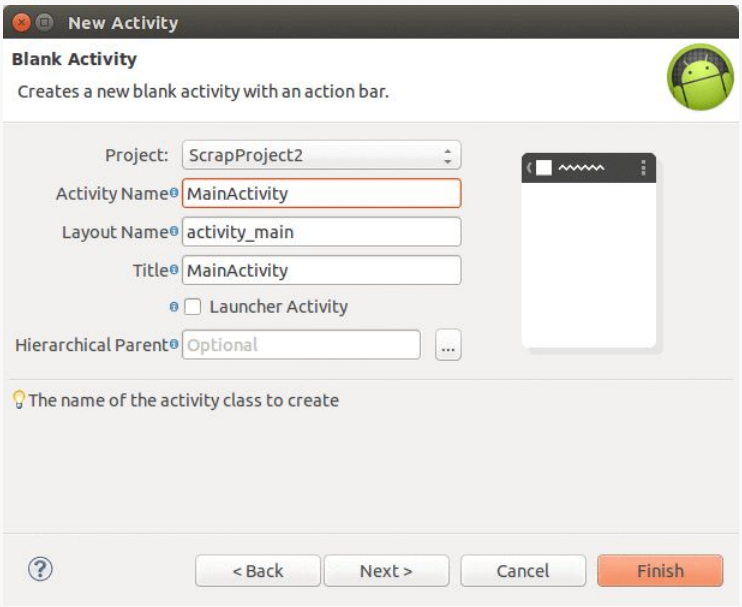


Figure 234: Eclipse New Activity Wizard, Second Page

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In addition to the class name and layout name, as we saw in the new-project wizard, we are also asked for some more details:

- the project into which we want this activity created (defaults to the one we right-clicked over)
- the "Title" is the caption that will appear in the action bar for this activity
- the "Launcher Activity" checkbox which, if checked, will add the appropriate bits to the manifest to tell Android that this activity should get an icon in the home screen's launcher
- the "Hierarchical Parent" drop-down, which presently is beyond the scope of this book

Clicking "Finish" will then create the activity's Java class, related resources, and manifest entry.

Populating the Class and Resources

Once you have your stub activity set up, you can then add an `onCreate()` method to it (or edit an existing one created by the wizard), filling in all the details (e.g., `setContentView()`), just like you did with your first activity. Your new activity may need a new layout XML resource or other resources, which you would also have to create (or edit those created for you by the wizard).

In `Activities/Explicit`, our second activity is `OtherActivity`, with pretty much the standard bare-bones implementation:

```
package com.commonsware.android.exint;

import android.app.Activity;
import android.os.Bundle;

public class OtherActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.other);
  }
}
```

and a similarly simple layout, `res/layout/other.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:orientation="vertical">

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/other"
    android:textColor="#FFFF0000"
    android:textSize="20sp"/>

</LinearLayout>
```

Augmenting the Manifest

Simply having an activity implementation is not enough. We also need to add it to our `AndroidManifest.xml` file. This is automatically handled for you by the IDEs' respective new-activity wizards. However, if you created the activity "by hand", you will need to add its manifest element, and over time you will need to edit this element in many cases.

Adding an activity to the manifest is a matter of adding another `<activity>` element to the `<application>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.exint"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="ExplicitIntentsDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
    <activity android:name="OtherActivity"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  </application>

</manifest>
```

You need the `android:name` attribute at minimum. Note that we do not include an `<intent-filter>` child element, the way the original activity has. For now, take it on faith that the original activity's `<intent-filter>` is what causes it to appear as a launchable activity in the home screen's launcher. We will get into more details of how that `<intent-filter>` works and when you might want your own in a later chapter.

Warning! Contains Explicit Intents!

An `Intent` encapsulates a request, made to Android, for some activity or other receiver to do something.

If the activity you intend to launch is one of your own, you may find it simplest to create an explicit `Intent`, naming the component you wish to launch. For example, from within your activity, you could create an `Intent` like this:

```
new Intent(this, HelpActivity.class);
```

This would stipulate that you wanted to launch the `HelpActivity`. This activity would need to be named in your `AndroidManifest.xml` file.

In `Activities/Explicit`, `ExplicitIntentsDemoActivity` has a `showOther()` method tied to its `Button` widget's `onClick` attribute. That method will use `startActivity()` with an explicit `Intent`, identifying `OtherActivity`:

```
package com.commonsware.android.exint;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class ExplicitIntentsDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void showOther(View v) {
    startActivity(new Intent(this, OtherActivity.class));
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }
}
```

Our launched activity shows the button:

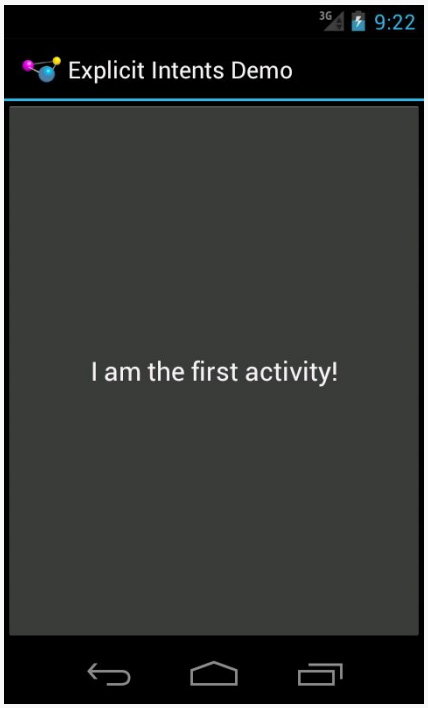


Figure 235: The Explicit Intents Demo, As Launched

Clicking the button brings up the other activity:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

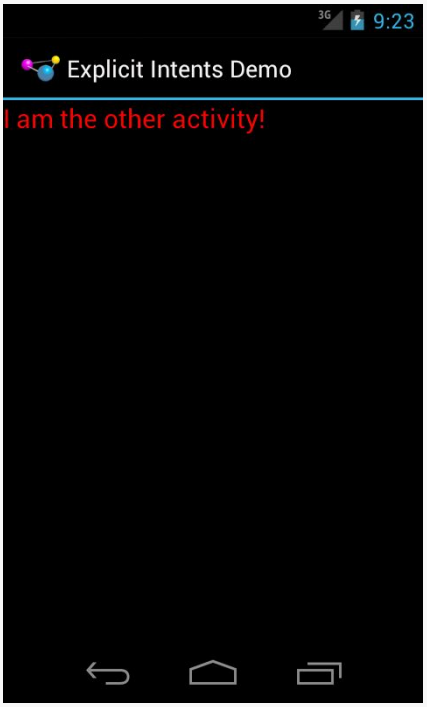


Figure 236: The Explicit Intents Demo, After Clicking the Button

Clicking BACK would return us to the first activity. In this respect, the BACK button in Android works much like the BACK button in your Web browser.

Using Implicit Intents

The explicit `Intent` approach works fine when the activity to be started is one of yours.

However, you can also start up activities from the operating system or third-party apps. In those cases, though, you will not have a Java `Class` object representing the other activity in your project, so you cannot use the `Intent` constructor that takes a `Class`.

Instead, you will use what are referred as the "implicit" `Intent` structure, which looks an *awful* lot like how the Web works.

If you have done any work on Web apps, you are aware that HTTP is based on verbs applied to URIs:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- We want to GET this image
- We want to POST to this script or controller
- We want to PUT to this REST resource
- Etc.

Android's implicit Intent model works much the same way, just with a *lot* more verbs.

For example, suppose you get a latitude and longitude from somewhere (e.g., body of a tweet, body of a text message). You decide that you want to display a map on those coordinates. There are ways that you can embed a Google Map directly in your app — and we will see how in a later chapter — but that is complicated and assumes the user wants Google Maps. It would be better if we could create some sort of generic "hey, Android, display an activity that shows a map for this location" request.

Or, in a simpler scenario: we get a URL to a Web page from some source (e.g., Web service call), and we want to open a Web browser on that page. This is illustrated in the Activities/LaunchWeb sample project.

We have a LaunchDemo activity that uses a layout containing a EditText widget and a Button, among other things:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <EditText
    android:id="@+id/url"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/url"
    android:inputType="textUri"/>

  <Button
    android:id="@+id/browse"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="showMe"
    android:text="@string/show_me"/>

</LinearLayout>
```

The Button is tied to a showMe() method on the activity itself, where we want to bring up a Web browser on the URL entered into the EditText widget:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.activities;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class LaunchDemo extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
  }

  public void showMe(View v) {
    EditText url=(EditText)findViewById(R.id.url);

    startActivity(new Intent(Intent.ACTION_VIEW,
                             Uri.parse(url.getText().toString())));
  }
}
```

Just as HTTP uses a verb and a URI, Android uses an action and a `Uri`. We take the URL and convert it to a `Uri` via calling `Uri.parse()`. Then, we can use an action called `ACTION_VIEW` to try to display the desired Web page.

When launched, the user is presented with our data entry form:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

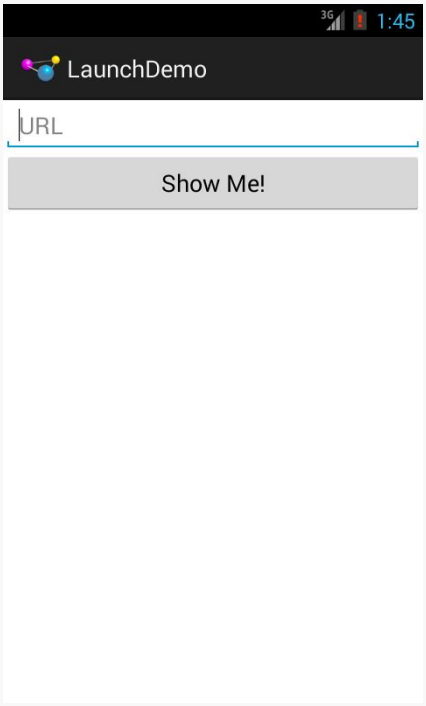


Figure 237: LaunchWeb Demo, As Initially Launched

We can fill in a URL:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

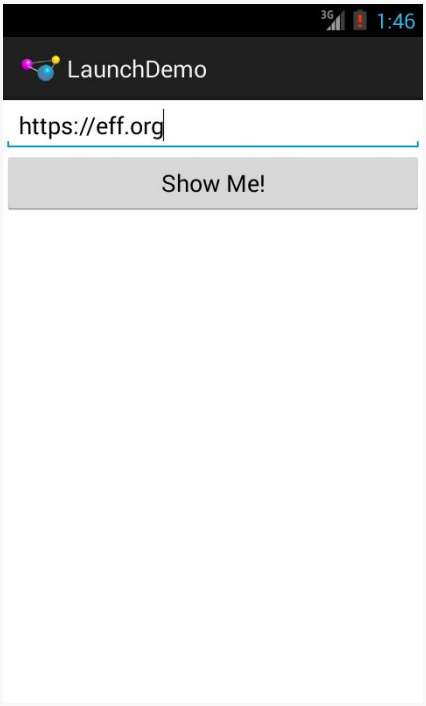


Figure 238: LaunchWeb Demo, After Data Entry

If the device has one app that responds to an `ACTION_VIEW` `Intent` on an `https:` scheme, clicking the "Show Me!" button will bring up that app, probably a Web browser:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

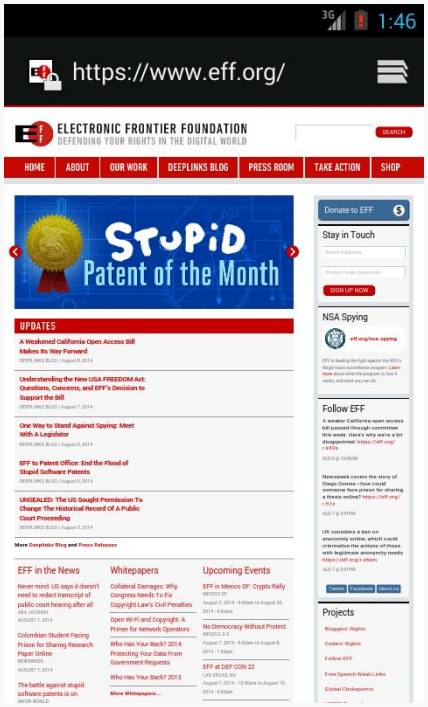


Figure 239: EFF Home Page, Launched from LaunchWeb

We will discuss what happens if there are no applications set up to handle this `Intent`, or if there is more than one, in a later chapter.

Extra! Extra!

Sometimes, we may wish to pass some data from one activity to the next. For example, we might have a `ListActivity` showing a collection of our model objects (e.g., books) and we have a separate `DetailActivity` to show information about a specific model object. Somehow, `DetailActivity` needs to know which model object to show.

One way to accomplish this is via `Intent` extras.

There is a series of `putExtra()` methods on `Intent` to allow you to supply key/value pairs of data to be bundled into the `Intent`. While you cannot pass arbitrary objects, most primitive data types are supported, as are strings and some types of lists. The next section will explain a bit more about what can go in an `Intent` extra.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Any activity can call `getIntent()` to retrieve the `Intent` used to start it up, and then can call various forms of `get... Extra()` (with the `...` indicating a data type) to retrieve any bundled extras.

For example, let's take a look at the `Activities/Extras` sample project.

This is mostly a clone of the `Activities/Explicit` sample from earlier in this chapter. However, this time, our first activity will pass an extra to the second:

```
package com.commonsware.android.extra;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class ExtrasDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void showOther(View v) {
    Intent other=new Intent(this, OtherActivity.class);

    other.putExtra(OtherActivity.EXTRA_MESSAGE, getString(R.string.other));
    startActivity(other);
  }
}
```

We create the `Intent` as before, but then call `putExtra()`, supplying a key (a static string named `OtherActivity.EXTRA_MESSAGE`) and a value (the `R.string.other` string resource). Then, and only then, do we call `startActivity()`.

Our revised `OtherActivity` then retrieves that extra, along with the inflated `TextView` (via `findViewById()`) and pours that text in:

```
package com.commonsware.android.extra;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class OtherActivity extends Activity {
  public static final String EXTRA_MESSAGE="msg";

  @Override
  public void onCreate(Bundle savedInstanceState) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onCreate(savedInstanceState);
    setContentView(R.layout.other);

    TextView tv=(TextView)findViewById(R.id.msg);

    tv.setText(getIntent().getStringExtra(EXTRA_MESSAGE));
  }
}
```

Visually, the result is the same. Functionally, the text to be shown is passed from one activity to the next.

Pondering Parcelable

As noted above, `Intent` extras cannot handle arbitrary objects. That is because, most of the time, `Intent` extras get passed across process boundaries. Even when you are calling `startActivity()` to start up one of your own activities, that request passes from your process to a core OS process and back to your process. The `Intent` extras come along for the ride.

Hence, `Intent` extras need to be something that can be converted into a `byte` array, as part of the inter-process communication (IPC) that handles the passing around of `Intent` objects. You will see this come up in other flavors of Android IPC as well, such as remote services.

However, there are two ways in which you can try to make your own objects work as `Intent` extras.

One approach is to implement `Serializable` on your class. This is a classic Java construct, designed to allow instances of your class, and other `Serializable` objects your instances hold onto, to be serialized into files and later read back in.

Another approach is to implement `Parcelable` on your class. This is an Android construct, one that is very similar to `Serializable`. However, `Serializable` is designed for *durable* storage of objects, where the file might be read back in months or years later. As such, `Serializable` has to deal with possible changes to the Java code implementing those classes, and as such needs to have hooks to help with converting old, saved objects into new objects. This adds overhead. `Parcelable` is only concerned with converting objects into `byte` arrays to pass across process boundaries. It can make the simplifying assumption that the class definition is not changing from when the object is turned into bytes and when the bytes are turned

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

back into an object. As a result, `Parcelable` is faster than `Serializable` for Android's IPC use.

You are welcome to implement `Parcelable` on your own classes if you wish, at which point they can be passed around via `Intent` extras. Beyond that, though, any Java classes you see in the Android JavaDocs that implement `Parcelable` can be put into `Intent` extras. So, for example, `Uri` implements `Parcelable`, and so you can put a `Uri` into an `Intent` extra. Not everything in the Android SDK is `Parcelable`, but some key classes like `Uri` are `Parcelable`.

Asynchronicity and Results

Note that `startActivity()` is asynchronous. The other activity will not show up until sometime after you return control of the main application thread to Android.

Normally, this is not much of a problem. However, sometimes one activity might start another, where the first activity would like to know some "results" from the second. For example, the second activity might be some sort of "chooser", to allow the user to pick a file or contact or song or something, and the first activity needs to know what the user chose. With `startActivity()` being asynchronous, it is clear that we are not going to get that sort of result as a return value from `startActivity()` itself.

To handle this scenario, there is a separate `startActivityForResult()` method. While it too is asynchronous, it allows the newly-started activity to supply a result (via a `setResult()` method) that is delivered to the original activity via an `onActivityResult()` method. We will examine `startActivityForResult()` in greater detail in a later chapter.

Schroedinger's Activity

An activity, generally speaking, is in one of four states at any point in time:

1. *Active*: the activity was started by the user, is running, and is in the foreground. This is what you are used to thinking of in terms of your activity's operation.
2. *Paused*: the activity was started by the user, is running, and is visible, but another activity is overlaying part of the screen. During this time, the user can see your activity but may not be able to interact with it. This is a

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

relatively uncommon state, as most activities are set to fill the screen, not have a theme that makes them look like some sort of dialog box.
3. *Stopped*: the activity was started by the user, is running, but it is hidden by other activities that have been launched or switched to.
4. *Dead*: the activity was destroyed, perhaps due to the user pressing the BACK button.

Life, Death, and Your Activity

Android will call into your activity as the activity transitions between the four states listed above.

Note that for all of these, you should chain upward and invoke the superclass' edition of the method, or Android may raise an exception.

`onCreate()` and `onDestroy()`

We have been implementing `onCreate()` in all of our `Activity` subclasses in all the examples. This will get called in two primary situations:

- When the activity is first started (e.g., since a system restart), `onCreate()` will be invoked with a `null` parameter.
- If the activity had been running and you have set up your activity to have different resources based on different device states (e.g., landscape versus portrait), your activity will be re-created and `onCreate()` will be called. We will discuss this scenario in greater detail later in this book.

Here is where you initialize your user interface and set up anything that needs to be done once, regardless of how the activity gets used.

On the other end of the lifecycle, `onDestroy()` may be called when the activity is shutting down, such as because the activity called `finish()` (which "finishes" the activity) or the user presses the BACK button. Hence, `onDestroy()` is mostly for cleanly releasing resources you obtained in `onCreate()` (if any), plus making sure that anything you started up outside of lifecycle methods gets stopped, such as background threads.

Bear in mind, though, that `onDestroy()` may not be called. This would occur in a few circumstances:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- You crash with an unhandled exception
- The user force-stops your application, such as through the Settings app
- Android has an urgent need to free up RAM (e.g., to handle an incoming phone call), wants to terminate your process, and cannot take the time to call all the lifecycle methods

Hence, `onDestroy()` is very likely to be called, but it is not guaranteed.

Also, bear in mind that it may take a long time for `onDestroy()` to be called. It is called quickly if the user presses BACK to finish the foreground activity. If, however, the user presses HOME to bring up the home screen, your activity is not immediately destroyed. `onDestroy()` will not be called until Android does decide to gracefully terminate your process, and that could be seconds, minutes, or hours later.

`onStart()`, `onRestart()`, and `onStop()`

An activity can come to the foreground either because it is first being launched, or because it is being brought back to the foreground after having been hidden (e.g., by another activity, by an incoming phone call).

The `onStart()` method is called in either of those cases. The `onRestart()` method is called in the case where the activity had been stopped and is now restarting.

Conversely, `onStop()` is called when the activity is about to be stopped. It too may not be called, for the same reasons that `onDestroy()` would not be called. However, `onStop()` is usually called fairly quickly after the activity is no longer visible, so the odds that `onStop()` will be called are even higher than that of `onDestroy()`.

`onPause()` and `onResume()`

The `onResume()` method is called just before your activity comes to the foreground, either after being initially launched, being restarted from a stopped state, or after a pop-up dialog (e.g., incoming call) is cleared. This is a great place to refresh the UI based on things that may have occurred since the user last was looking at your activity. For example, if you are polling a service for changes to some information (e.g., new entries for a feed), `onResume()` is a fine time to both refresh the current view and, if applicable, kick off a background thread to update the view (e.g., via a `Handler`).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Conversely, anything that takes over user input — mostly, the activation of another activity — will result in your `onPause()` being called. Here, you should undo anything you did in `onResume()`, such as stopping background threads, releasing any exclusive-access resources you may have acquired (e.g., camera), and the like.

Once `onPause()` is called, Android reserves the right to kill off your activity's process at any point. Hence, you should not be relying upon receiving any further events.

So, what is the difference between `onPause()` and `onStop()`? If an activity comes to the foreground that fills the screen, your current foreground activity will be called with `onPause()` and `onStop()`. If, however, an activity comes to the foreground that does *not* fill the screen, your current foreground activity will only be called with `onPause()`, as it is still visible.

Stick to the Pairs

If you initialize something in `onCreate()`, clean it up in `onDestroy()`.

If you initialize something in `onStart()`, clean it up in `onStop()`.

If you initialize something in `onResume()`, clean it up in `onPause()`.

In other words, stick to the pairs. For example, do not initialize something in `onStart()` and try to clean it up on `onPause()`, as there are scenarios where `onPause()` may be called multiple times in succession (i.e., user brings up a non-full-screen activity, which triggers `onPause()` but not `onStop()`, and hence not `onStart()`).

Which pairs of lifecycle methods you choose is up to you, depending upon your needs. You may decide that you need two pairs (e.g., `onCreate()`/`onDestroy()` and `onResume()`/`onPause()`). Just do not mix and match between them.

When Activities Die

So, what gets rid of an activity? What can trigger the chain of events that results in `onDestroy()` being called?

First and foremost, when the user presses the BACK button, the foreground activity will be destroyed, and control will return to the previous activity in the user's

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

navigation flow (i.e., whatever activity they were on before the now-destroyed activity came to the foreground).

You can accomplish the same thing by calling `finish()` from your activity. This is mostly for cases where some other UI action would indicate that the user is done with the activity (e.g., the activity presents a list for the user to choose from — clicking on a list item might close the activity). However, please do not artificially add your own "exit", "quit", or other menu items or buttons to your activity — just allow the user to use normal Android navigation options, such as the BACK button.

If none of your activities are in the foreground any more, your application's process is a candidate to be terminated to free up RAM. As noted earlier, depending on circumstances, Android may or may not call `onDestroy()` in these cases (`onPause()` and `onStop()` would have been called when your activities left the foreground).

If the user causes the device to go through a "configuration change", such as switching between portrait and landscape, Android's default behavior is to destroy your current foreground activity and create a brand new one in its place. We will cover this more in a later chapter.

And, if your activity has an unhandled exception, your activity will be destroyed, though Android will not call any more lifecycle methods on it, as it assumes your activity is in an unstable state.

Walking Through the Lifecycle

To see when these various lifecycle methods get called, let's examine the `Activities/Lifecycle` sample project.

This project is the same as the `Activities/Extras` project, except that our two activities no longer inherit from `Activity` directly. Instead, we introduce a `LifecycleLoggingActivity` as a base class and have our activities inherit from it:

```
package com.commonsware.android.lifecycle;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

public class LifecycleLoggingActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    Log.d(getClass().getSimpleName(), "onCreate()");
  }

  @Override
  public void onRestart() {
    super.onRestart();

    Log.d(getClass().getSimpleName(), "onRestart()");
  }

  @Override
  public void onStart() {
    super.onStart();

    Log.d(getClass().getSimpleName(), "onStart()");
  }

  @Override
  public void onResume() {
    super.onResume();

    Log.d(getClass().getSimpleName(), "onResume()");
  }

  @Override
  public void onPause() {
    Log.d(getClass().getSimpleName(), "onPause()");

    super.onPause();
  }

  @Override
  public void onStop() {
    Log.d(getClass().getSimpleName(), "onStop()");

    super.onStop();
  }

  @Override
  public void onDestroy() {
    Log.d(getClass().getSimpleName(), "onDestroy()");

    super.onDestroy();
  }
}
```

All `LifecycleLoggingActivity` does is override each of the lifecycle methods mentioned above and emit a debug line to LogCat indicating who called what.

When we first launch the application, our first batch of lifecycle methods is invoked, in the expected order:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
04-01 11:47:21.437: D/ExplicitIntentsDemoActivity(1473): onCreate()
04-01 11:47:21.827: D/ExplicitIntentsDemoActivity(1473): onStart()
04-01 11:47:21.827: D/ExplicitIntentsDemoActivity(1473): onResume()
```

If we click the button on the first activity to start up the second, we get:

```
04-01 11:47:54.776: D/ExplicitIntentsDemoActivity(1473): onPause()
04-01 11:47:54.877: D/OtherActivity(1473): onCreate()
04-01 11:47:54.947: D/OtherActivity(1473): onStart()
04-01 11:47:54.974: D/OtherActivity(1473): onResume()
04-01 11:47:55.347: D/ExplicitIntentsDemoActivity(1473): onStop()
```

Notice that our first activity is paused before the second activity starts up, and that `onStop()` is delayed on the first activity until after the second activity has appeared.

If we press the BACK button on the second activity, returning to the first activity, we see:

```
04-01 11:48:54.807: D/OtherActivity(1473): onPause()
04-01 11:48:54.857: D/ExplicitIntentsDemoActivity(1473): onRestart()
04-01 11:48:54.857: D/ExplicitIntentsDemoActivity(1473): onStart()
04-01 11:48:54.857: D/ExplicitIntentsDemoActivity(1473): onResume()
04-01 11:48:55.257: D/OtherActivity(1473): onStop()
04-01 11:48:55.257: D/OtherActivity(1473): onDestroy()
```

Notice how, once again, going onto the screen happens in between `onPause()` and `onStop()` of the activity leaving the screen. Also notice that `onDestroy()` is called immediately after `onStop()`, because the activity was finished via the BACK button.

If we now press the HOME button, to bring the home screen activity to the foreground, we see:

```
04-01 11:50:30.347: D/ExplicitIntentsDemoActivity(1473): onPause()
04-01 11:50:32.227: D/ExplicitIntentsDemoActivity(1473): onStop()
```

There is a delay between `onPause()` and `onStop()` as the home screen does its display work, and there is no `onDestroy()`, because the application is still running

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

and nothing finished the activity. Eventually, the device will terminate our process, and if that happens normally, we would see the `onDestroy()` LogCat message.

Recycling Activities

Let us suppose that we have three activities, named A, B, and C. A starts up an instance of B based on some user input, and B later starts up an instance of C through some more user input.

Our "activity stack" is now A-B-C, meaning that if we press BACK from C, we return to B, and if we press BACK from B, we return to A.

Now, let's suppose that from C, we wish to navigate back to A. For example, perhaps the user pressed the icon on the left of our action bar, and we want to return to the "home activity" as a result, and in our case that happens to be A. If C calls `startActivity()`, specifying A, we wind up with an activity stack that is A-B-C-A.

That's because starting an activity, by default, creates a new instance of that activity. So, now we have two independent copies of A.

Sometimes, this is desired behavior. For example, we might have a single `ListActivity` that is being used to "drill down" through a hierarchical data set, like a directory tree. We might elect to keep starting instances of that same `ListActivity`, but with different extras, to show each level of that hierarchy. In this case, we would want independent instances of the activity, so the BACK button behaves as the user might expect.

However, when we navigate to the "home activity", we may not want a separate instance of A.

How to address this depends a bit on what you want the activity stack to look like after navigating to A.

If you want an activity stack that is B-C-A — so the existing copy of A is brought to the foreground, but the instances of B and C are left alone — then you can add `FLAG_ACTIVITY_REORDER_TO_FRONT` to your `Intent` used with `startActivity()`:

```
Intent i=new Intent(this, HomeActivity.class);

i.setFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT);
startActivity(i);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If, instead, you want an activity stack that is just A — so if the user presses BACK, they exit your application — then you would add two flags: `FLAG_ACTIVITY_CLEAR_TOP` and `FLAG_ACTIVITY_SINGLE_TOP`:

```
Intent i=new Intent(this, HomeActivity.class);

i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.FLAG_ACTIVITY_SINGLE_TOP);
startActivity(i);
```

This will finish all activities in the stack between the current activity and the one you are starting — in our case, finishing C and B.

Application: Transcending the Activity

`Activity` inherits from a class named `Context`. Many of the methods that we are calling on our activities, like `startActivity()`, are inherited from `Context`.

However, `Activity` is not the only relevant subclass of `Context`. We will see `Service` later in the book, for example. And sometimes we will see plain `Context` objects, such as when we cover `BroadcastReceiver` later in the book.

Another `Context` of note is `Application`. An instance of `Application` is created when our app starts up. The `Application` instance is a natural singleton; there should be exactly one instance of `Application` in our process.

Normally, this singleton is an instance of `Application` itself. However, we can subclass `Application` if we wish, then include a reference to our custom application class in an `android:name` attribute on the `<application>` element in the manifest. Then, when Android starts up our app, it will create an instance of our designated `Application` subclass, rather than creating an instance of the ordinary `Application` class.

We can retrieve the `Application` object at any point by calling `getApplicationContext()` on any `Context` object. `getApplicationContext()` will return a `Context`; if we need to reference `Application` or our specific `Application` subclass, we need to down-cast the returned `Context` to the appropriate type.

We can use `Application` in a few ways in Android apps.

First and foremost, if we need to hold onto some other object in a `static` data member, and that other object needs a `Context`, we *really* want it to be using the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Application, not an Activity, Service, etc. Because Application is a singleton, it is effectively "pre-leaked". We cannot somehow leak it further by having another indirect static reference to it. In contrast, suppose we have a static data member holding onto an Activity. Now, when that Activity is destroyed, it (and all it holds, like widgets and listeners) cannot be garbage-collected. This represents a memory leak.

You could even take it one step further and have the Application manage this static data, rather than using separate singletons. There are pros and cons to this approach, but on the whole Google is not a big fan of it. That being said, Application has an onCreate() that is called shortly after it is instantiated, and your subclass of Application could override that and use it to initialize some "global" data.

However, while the JavaDocs indicate that there is an onTerminate() method on Application — suggesting that we find out when the Application is going away and our process is being terminated — that method is never called in practice.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #8 - Setting Up An Activity

Of course, it would be nice if those "Help" and "About" menu choices that we added in the previous tutorial actually did something.

In this tutorial, we will define another activity class, one that will be responsible for displaying simple content like our help text and "about" details. And, we will arrange to start up that activity when those action bar items are selected. The activity will not actually display anything meaningful yet, as that will be the subject of the next few tutorials.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Creating the Stub Activity Class and Manifest Entry

First, we need to define the Java class for our new activity, `SimpleContentActivity`.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio

Right-click on your `main/` sourceset directory in the project explorer, and choose New > Activity > Blank Activity from the context menu. This will bring up a new-activity wizard:

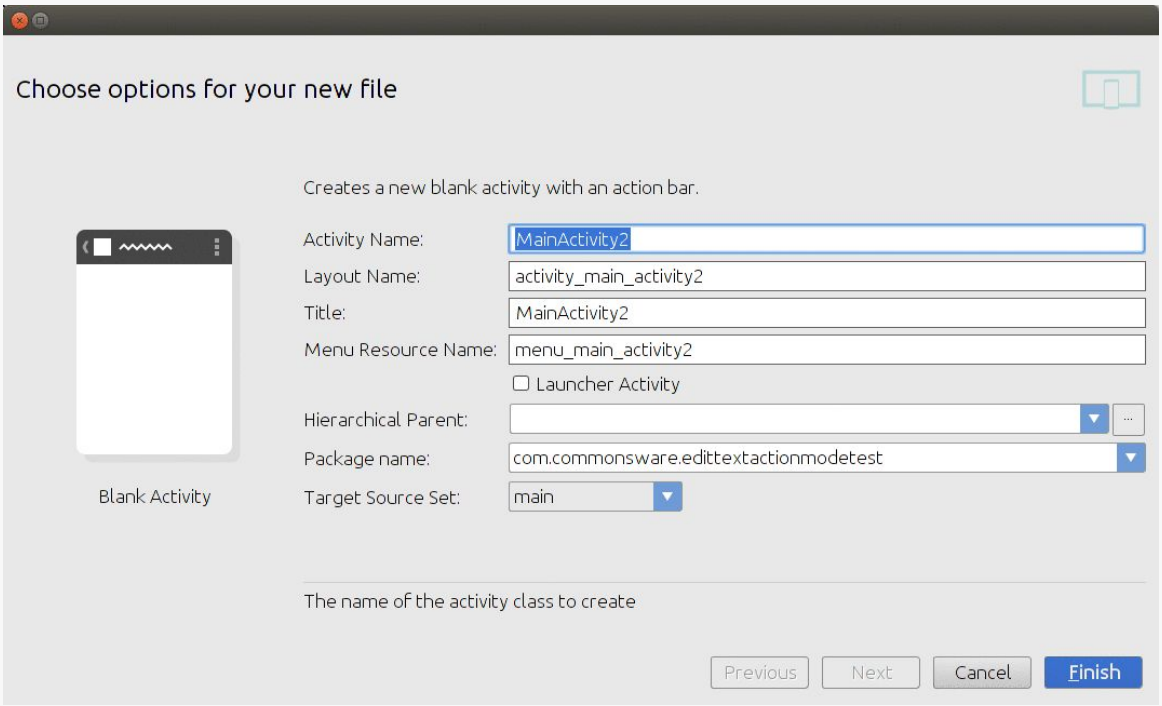


Figure 240: Android Studio New Activity Wizard

Fill in:

- `SimpleContentActivity` in the "Activity Name" field
- `pleasedeleteme` for the "Layout Name"
- `@string/app_name` for the "Title"
- `pleasedeletemetoo` for the "Menu Resource Name"

Leave the rest of the wizard at its defaults, then click on Finish.

In the `SimpleContentActivity` class that results, delete:

- the `onCreateOptionsMenu()` method
- the `onOptionsItemSelected()` method
- the `setContentView()` call from the `onCreate()` method

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

At this point, your `SimpleContentActivity` class should look like:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.os.Bundle;

public class SimpleContentActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

At this point, you can delete the `res/layout/pleasedeleteme.xml` file, by right-clicking over it and choosing "Delete..." from the context menu. You can also delete the `res/menu/simple_content.xml` file, the `res/values/dimens.xml` file, the `res/values-820dp/` directory, and the `res/menu/pleasedeletemetoo.xml` file.

Next, go into `AndroidManifest.xml` and delete the `android:label` attribute of the `SimpleContentActivity` `<activity>` element.

Finally, you can open up the `res/values/strings.xml` file and delete the following string resources:

- `title_activity_simple_content`
- `hello_world`
- `action_settings`

Eclipse

Right-click over the EmPubLite project in the Package Explorer, then choose New > Other from the context menu. In the list of possible things to add to your project, choose "Android Activity" and click Next. In the list of templates, choose "Empty Activity" and click Next.

In the last page of the new-activity wizard, fill in:

- `SimpleContentActivity` in the "Activity Name" field
- `pleasedeleteme` for the "Layout Name"
- `pleasedeletemetoo` for the "Title"

Leave the rest of the wizard at its defaults, then click on Finish.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In the `SimpleContentActivity` class that Android Studio generates, delete the `setContentView()` call from the `onCreate()` method.

You can also delete the `/res/layout/pleasedeleteme.xml` file, by right-clicking over it and choosing "Delete..." from the context menu.

Then, open up `AndroidManifest.xml`. In the XML sub-tab, go to the `<activity>` element added for `SimpleContentActivity` and remove the `android:label` attribute, so we inherit our label from the `<application>` element. At this point, you can save your changes to this file (e.g., `<Ctrl>-<S>`).

Finally, open up `res/values/strings.xml` and delete the `title_activity_simple_content` that the wizard added, which we are no longer using. At this point, you can save your changes to this file (e.g., `<Ctrl>-<S>`).

Step #2: Launching Our Activity

Now that we have declared that the activity exists and can be used, we can start using it.

Go into `EmPubLiteActivity` and modify `onOptionsItemSelected()` to add in some logic in the `R.id.about` and `R.id.help` branches, as shown below:

```
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case android.R.id.home:
                return(true);

            case R.id.about:
                Intent i=new Intent(this, SimpleContentActivity.class);
                startActivity(i);

                return(true);

            case R.id.help:
                i=new Intent(this, SimpleContentActivity.class);
                startActivity(i);

                return(true);
        }
        return(super.onOptionsItemSelected(item));
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In those two branches, we create an `Intent`, pointing at our new `SimpleContentActivity`. Then, we call `startActivity()` on that `Intent`. Right now, both `help` and `about` do the same thing — we will add some smarts to have them load up different content later in this book.

You will need to add an import for `android.content.Intent` to get this to compile.

If you run this app in a device or emulator, and you choose either the Help or About menu choices, what appears to happen is that the `ProgressBar` vanishes. In reality, what happens is that our `SimpleContentActivity` appeared, but empty, as we have not given it a full UI yet.

In Our Next Episode…

… we will begin using fragments in our tutorial project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Tactics of Fragments

Fragments are an optional layer you can put between your activities and your widgets, designed to help you reconfigure your activities to support screens both large (e.g., tablets) and small (e.g., phones).

This chapter will cover basic uses of fragments.

The Six Questions

In the world of journalism, the basics of any news story consist of six questions, the Five Ws and One H. Here, we will apply those six questions to help frame what we are talking about with respect to fragments.

What?

Fragments are not activities, though they can be used by activities.

Fragments are not containers (i.e., subclasses of `ViewGroup`), though typically they create a `ViewGroup`.

Rather, you should think of fragments as being units of UI reuse. You define a fragment, much like you might define an activity, with layouts and lifecycle methods and so on. However, you can then host that fragment in one or several activities, as needed.

Android does not precisely implement UI architectures like Model-View-Controller (MVC), Model-View-Presenter (MVP), Model-View-ViewModel (MVVM), etc. To the extent that you wish to shove Android into the MVC architecture, fragments and activities combine to be the controller layer. Fragments serve as a local controller,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

focused on their set of widgets, populating them from model data, and handling their events. Activities will serve as more of an orchestration layer, handling cross-fragment communications (e.g., a click in Fragment A needs to cause a change in what is displayed in Fragment B).

Functionally, fragments are Java classes, extending from a base `Fragment` class. As we will see, there are two versions of the `Fragment` class, one native to API Level 11 and one supplied by the Android Support package.

Where??

Since fragments are Java classes, your fragments will reside in one of your application's Java packages. The simplest approach is to put them in the same Java package that you used for your project overall and where your activities reside, though you can refactor your UI logic into other packages if needed.

Who?!?

Typically, you create fragment implementations yourself, then tell Android when to use them. Some third-party Android library projects may ship fragment implementations that you can reuse, if you so choose.

When?!!?

Some developers start adding fragments from close to the outset of application development — that is the approach we will take in the tutorials. And, if you are starting a new application from scratch, defining fragments early on is probably a good idea. That being said, it is entirely possible to "retrofit" an existing Android application to use fragments, though this may be a lot of work. And, it is entirely possible to create Android applications without fragments at all.

Fragments were introduced with Android 3.0 (API Level 11, a.k.a., Honeycomb).

WHY?!?!?

Ah, this is the big question. If we have managed to make it this far through the book without fragments, and we do not necessarily need fragments to create Android applications, what is the point? Why would we bother?

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The primary rationale for fragments was to make it easier to support multiple screen sizes.

Android started out supporting phones. Phones may vary in size, from tiny ones with less than 3" diagonal screen size (e.g., Sony Ericsson X10 mini), to monsters that are over 5" (e.g., Samsung Galaxy Note). However, those variations in screen size pale in comparison to the differences between phones and tablets, or phones and TVs.

Some applications will simply expand to fill larger screen sizes. Many games will take this approach, simply providing the user with bigger interactive elements, bigger game boards, etc. Any one of the ever-popular Angry Birds game series, when played on an tablet, gives you bigger birds and bigger pigs, not a phone-sized game area surrounded by ad banners.

However, another design approach is to consider a tablet screen to really be a collection of phone screens, side by side.

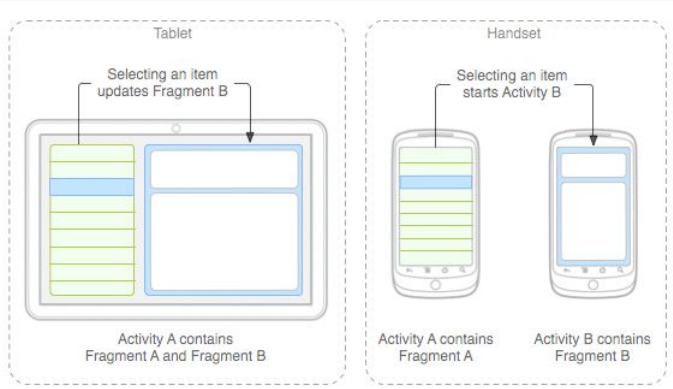


Figure 241: Tablets vs. Handsets (image courtesy of Android Open Source Project)

The user can access all of that functionality at once on a tablet, whereas they would have to flip back and forth between separate screens on a phone.

For applications that can fit this design pattern, fragments allow you to support phones and tablets from one code base. The fragments can be used by individual activities on a phone, or they can be stitched together by a single activity for a tablet.

Details on using fragments to support large screen sizes is a topic for a later chapter in this book. This chapter is focused on the basic mechanics of setting up and using fragments.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

OMGOMGOMG, HOW?!?!??

Well, answering that question is what the rest of this chapter is for, plus coverage of more advanced uses of fragments elsewhere in this book.

Where You Get Your Fragments From

Most developers will use the implementation of fragments that has been part of Android since API Level 11. You will use `android.app.Fragment` and have them be hosted by your regular activities.

However, there is a backport of the fragment system available in the Android Support package. This works back to API Level 4, so if your `minSdkVersion` is lower than 11 *and* you want to use fragments, the backport is something that you will wish to consider. You will need to add the `support-v4` JAR to your project (e.g., via `compile 'com.android.support:support-v4:...'` in your `dependencies` in `build.gradle`, for some value of `...`). You will also need to use `android.support.v4.app.Fragment` instead of `android.app.Fragment`. Also, you will need to host those fragments in an activity inheriting from `android.support.v4.app.FragmentActivity` (as the regular `android.app.Activity` base class does not know about fragments prior to API Level 11).

This book focuses mostly on using the native API Level 11 implementation of fragments, with the occasional example of using the backport where the backport is necessary for one reason or another.

Your First Fragment

In many ways, it is easier to explain fragments by looking at an implementation, more so than trying to discuss them as abstract concepts. So, in this section, we will take a look at the `Fragments/Static` sample project. This is a near-clone of the `Activities/Lifecycle` sample project from the previous chapter. However, we have converted the launcher activity from one that will host widgets directly itself to one that will host a fragment, which in turn manages widgets.

The Fragment Layout

Our fragment is going to manage our UI, so we have a `res/layout/mainfrag.xml` layout file containing our `Button`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/showOther"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:text="@string/hello"
  android:textSize="20sp"/>
```

Note, though, that we do not use the `android:onClick` attribute. We will explain why we dropped that attribute from the previous editions of this sample shortly.

The Fragment Class

The project has a `ContentFragment` class that will use this layout and handle the `Button`.

As with activities, there is no constructor on a typical `Fragment` subclass. The primary method you override, though, is not `onCreate()` (though, as we will see later in this chapter, that is possible). Instead, the primary method to override is `onCreateView()`, which is responsible for returning the UI to be displayed for this fragment:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.mainfrag, container, false);

    result.findViewById(R.id.showOther).setOnClickListener(this);

    return(result);
  }
```

We are passed a `LayoutInflater` that we can use for inflating a layout file, the `ViewGroup` that will eventually hold anything we inflate, and the `Bundle` that was passed to the activity's `onCreate()` method. While we are used to framework classes loading our layout resources for us, we can "inflate" a layout resource at any time using a `LayoutInflater`. This process reads in the XML, parses it, walks the element tree, creates Java objects for each of the elements, and stitches the results together into a parent-child relationship.

Here, we inflate `res/layout/mainfrag.xml`, telling Android that its contents will eventually go into the `ViewGroup` but not to add it right away. While there are simpler flavors of the `inflate()` method on `LayoutInflater`, this one is required in

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

case the ViewGroup happens to be a RelativeLayout, so we can process all of the positioning and sizing rules appropriately.

We also use findViewById() to find our Button widget and tell it that we, the fragment, are its OnClickListener. ContentFragment must then implement the View.OnClickListener interface to make this work. We do this instead of android:onClick to route the Button click events to the fragment, not the activity.

Since we implement the View.OnClickListener interface, we need the corresponding onClick() method implementation:

```
  @Override
  public void onClick(View v) {
    ((StaticFragmentsDemoActivity)getActivity()).showOther(v);
  }
```

Any fragment can call getActivity() to find the activity that hosts it. In our case, the only activity that will possibly host this fragment is StaticFragmentsDemoActivity, so we can cast the result of getActivity() to StaticFragmentsDemoActivity, so that we can call methods on our activity. In particular, we are telling the activity to show the other activity, by means of calling the showOther() method that we saw in the original Activities/Lifecycle sample (and will see again shortly).

That is really all that is needed for this fragment. However, ContentFragment also overrides many other fragment lifecycle methods, and we will examine these later in this chapter.

The Activity Layout

Originally, the res/layout/main.xml used by the activity was where we had our Button widget. Now, the Button is handled by the fragment. Instead, our activity layout needs to account for the fragment itself.

In this sample, we are going to use a static fragment. Static fragments are easy to add to your application: just use the <fragment> element in a layout file, such as our revised res/layout/main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:name="com.commonsware.android.sfrag.ContentFragment"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here, we are declaring our UI to be completely comprised of one fragment, whose implementation (`com.commonsware.android.sfrag.ContentFragment`) is identified by the `android:name` attribute on the `<fragment>` element. Instead of `android:name`, you can use `class`, though most of the Android documentation has now switched over to `android:name`.

Android Studio users can drag a fragment out of the "Custom" section of the graphical layout editor tool palette, if desired, rather than setting up the `<fragment>` element directly in the XML. Eclipse users have a similar option in the "Layouts" section of the tool palette.

The Activity Class

`StaticFragmentsDemoActivity` — our new launcher activity — looks identical to the previous version, with the exception of the class name:

```
package com.commonsware.android.sfrag;

import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class StaticFragmentsDemoActivity extends
    LifecycleLoggingActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void showOther(View v) {
    Intent other=new Intent(this, OtherActivity.class);

    other.putExtra(OtherActivity.EXTRA_MESSAGE,
                   getString(R.string.other));
    startActivity(other);
  }
}
```

Since the `res/layout/main.xml` file has the `<fragment>` element, the fragment is simply loaded into position in the call to `setContentView()`.

The Fragment Lifecycle Methods

Fragments have lifecycle methods, just like activities do. In fact, they support most of the same lifecycle methods as activities:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `onCreate()`
- `onStart()` (but not `onRestart()`)
- `onResume()`
- `onPause()`
- `onStop()`
- `onDestroy()`

By and large, the same rules apply for fragments as do for activities with respect to these lifecycle methods (e.g., `onDestroy()` may not be called).

In addition to those and the `onCreateView()` method we examined earlier in this chapter, there are four other lifecycle methods that you can elect to override if you so choose.

`onAttach()` will be called first, even before `onCreate()`, letting you know that your fragment has been attached to an activity. You are passed the `Activity` that will host your fragment.

`onViewCreated()` will be called after `onCreateView()`. This is particularly useful if you are inheriting the `onCreateView()` implementation but need to configure the resulting views, such as with a `ListFragment` and needing to set up an adapter.

`onActivityCreated()` will be called after `onCreate()` and `onCreateView()`, to indicate that the activity's `onCreate()` has completed. If there is something that you need to initialize in your fragment that depends upon the activity's `onCreate()` having completed its work, you can use `onActivityCreated()` for that initialization work.

`onDestroyView()` is called before `onDestroy()`. This is the counterpart to `onCreateView()` where you set up your UI. If there are things that you need to clean up specific to your UI, you might put that logic in `onDestroyView()`.

`onDetach()` is called after `onDestroy()`, to let you know that your fragment has been disassociated from its hosting activity.

Your First Dynamic Fragment

Static fragments are fairly simple, once you have the `Fragment` implementation: just add the `<fragment>` element to where you want to have the fragment appear in your activity's layout.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That simplicity, though, does come with some costs. We will review some of those limitations in the next chapter.

Those limitations can be overcome by the use of dynamic fragments. Rather than indicating to Android that you wish to use a fragment by means of a `<fragment>` element in a layout, you will use a `FragmentTransaction` to add a fragment at runtime from your Java code.

With that in mind, take a look at the `Fragments/Dynamic` sample project.

This is the same project as the one for static fragments, except this time we will adjust `OtherActivity` to use a dynamic fragment, specifically a `ListFragment`.

The ListFragment Class

`ListFragment` serves the same role for fragments as `ListActivity` does for activities. It wraps up a `ListView` for convenient use. So, to have a more interesting `OtherActivity`, we start with an `OtherFragment` that is a `ListFragment`, designed to show our favorite 25 Latin words as seen in previous examples.

Just as a `ListActivity` does not need to call `setContentView()`, a `ListFragment` does not need to override `onCreateView()`. By default, the entire fragment will be comprised of a single `ListView`. And just as `ListActivity` has a `setListAdapter()` method to associate an `Adapter` with the `ListView`, so too does `ListFragment`:

```
package com.commonsware.android.dfrag;

import android.app.Activity;
import android.app.ListFragment;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.ArrayAdapter;

public class OtherFragment extends ListFragment {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };

  @Override
  public void onViewCreated(View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    setListAdapter(new ArrayAdapter<String>(getActivity(),
                   android.R.layout.simple_list_item_1, items));
```

We call `setListAdapter()` in `onViewCreated()`, as we know that the `ListView` is now ready for use.

This class also overrides many fragment lifecycle methods, logging their results, akin to our other `Fragment` and `LifecycleLoggingActivity`.

The Activity Class

Now, `OtherActivity` no longer needs to load a layout — we have removed `res/layout/other.xml` from the project entirely. Instead, we will use a `FragmentTransaction` to add our fragment to the UI:

```
package com.commonsware.android.dfrag;

import android.os.Bundle;

public class OtherActivity extends LifecycleLoggingActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new OtherFragment()).commit();
    }
  }
}
```

To work with a `FragmentTransaction`, you need the `FragmentManager`. This object knows about all of the fragments that exist in your activity. If you are using the native API Level 11 edition of fragments, you can get your `FragmentManager` by calling `getFragmentManager()`. If you are using the Android Support package, you need to call `getSupportFragmentManager()` instead.

Given a `FragmentManager`, you can start a `FragmentTransaction` by calling `beginTransaction()`, which returns the `FragmentTransaction` object. `FragmentTransaction` operates on the builder pattern, so most methods on `FragmentTransaction` return the `FragmentTransaction` itself, so you can chain a series of method calls one after the next.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We call two methods on our `FragmentTransaction`: `add()` and `commit()`. The `add()` method, as you might guess, indicates that we want to add a fragment to the UI. We supply the actual fragment object, in this case by creating a new `OtherFragment`. We also need to indicate where in our layout we want this fragment to reside. Had we loaded a layout, we could drop this fragment in any desired container. In our case, since we did not load a layout, we supply `android.R.id.content` as the ID of the container to hold our fragment's `View`. Here, `android.R.id.content` identifies the container into which the results of `setContentView()` would go — it is a container supplied by `Activity` itself and serves as the top-most container for our content.

Just calling `add()` is insufficient. We then need to call `commit()` to make the transaction actually happen.

You might be wondering why we are trying to find a fragment in our `FragmentManager` before actually creating the fragment. We do that to help deal with configuration changes, and we will be exploring that further in the next chapter.

Fragments and the Action Bar

Fragments can add items to the action bar by calling `setHasOptionsMenu(true)` from `onCreate()` (or any other early lifecycle method). This indicates to the activity that it needs to call `onCreateOptionsMenu()` and `onOptionsItemSelected()` on the fragment.

The `Fragments/ActionBarNative` sample application demonstrates this. This has the same functionality as does the `ActionBar/ActionBarDemoNative` sample from the chapter on the action bar, just with the activity converted into a dynamic fragment.

In `onCreate()`, we call `setHasOptionsMenu(true)`, to indicate that we are interested in participating in the action bar:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);
  setHasOptionsMenu(true);
}
```

(we will discuss that `setRetainInstance(true)` call in a later chapter)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That will trigger our fragment's `onCreateOptionsMenu()` and `onOptionsItemSelected()` methods to be called at the appropriate time:

```
  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.actions, menu);

    super.onCreateOptionsMenu(menu, inflater);
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
      case R.id.add:
        addWord();

        return(true);

      case R.id.reset:
        initAdapter();

        return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

Here, we initialize our action bar from the `R.menu.actions` menu XML resource, including setting up our `EditText` widget, plus the logic to respond to the `reset` action overflow item.

Our activity does not need to do anything special to allow the fragment to contribute to the action bar — it just sets up the dynamic fragment:

```
package com.commonsware.android.abf;

import android.app.Activity;
import android.os.Bundle;

public class ActionBarFragmentActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new ActionBarFragment()).commit();
    }
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Fragments Within Fragments: Just Say "Maybe"

Historically, one major limitation with fragments is that they could not contain other fragments. In most cases, this does not pose a major problem. However, there will be times when you might trip over this limitation, such as when using a `ViewPager`, as will be described in a later chapter.

Android 4.2 — and the Android Support package – added support for nested fragments. Whereas an activity works with fragments via a `FragmentManager` obtained via `getFragmentManager()` or `getSupportFragmentManager()`, fragments can work with nested fragments via a call to `getChildFragmentManager()`.

However, Android 3.0 through 4.1 have a version of fragments that does not have `getChildFragmentManager()`. Hence, you have two options:

1. Use the Android Support package's backport of fragments, until such time as you can drop support for Android 4.1 and earlier, or
2. Do not use nested fragments for the time being.

We will see how `getChildFragmentManager()` works in the chapter on `ViewPager`.

Fragments and Multiple Activities

A fragment should handle functionality purely within the fragment itself. Anything outside the fragment should be the responsibility of the calling activity. For example, if the user taps on an item in a `ListFragment`, and the effects of that event might go beyond what is inside the `ListFragment` itself, the `ListFragment` should forward the event to the hosting activity, so it can perhaps perform additional steps (e.g., launch an activity, update another fragment hosted by the activity).

As we will see in a later chapter, it is entirely possible — perhaps even likely — that some of our fragments will be hosted by multiple different activities. For example, we might have a fragment that is hosted in one case by an activity designed for larger screens (e.g., tablets) and in another case by an activity designed for smaller screens (e.g., phones).

In these cases, the fragment does not know at compile time which activity class will be hosting it at runtime. For those cases, you have two major options:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. Have the activities implement a common interface, and have the fragment cast the result of calling `getActivity()` to that interface, so it can call methods on the hosting activity without knowing its exact implementation.
2. Have the activities supply a listener object, with a common interface, to the fragment via a setter, and have the fragment use that listener for raising events and so on.
3. Use an event bus, as we will explore later in this book.

We will see much more on this subject when we get into large-screen strategies in a later chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #9 - Starting Our Fragments

Much of the content of a digital book to be viewed in EmPubLite will be in the form of HTML and related assets (CSS, images, etc.). Hence, we will eventually need to render our content in a `WebView` widget, for best results with semi-arbitrary HTML content.

To do this, we will set up fragments for the bits of content:

- each chapter (or, in our case, HTML file containing chapters)
- other material, like our "help" and "about" pages

Right now, we will focus on just setting up some of the basic classes for these fragments — we will load them up with content and display them over the next few tutorials.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Create a SimpleContentFragment

Android has a `WebViewFragment` for the native API Level 11+ implementation of fragments, designed to show some Web content in a `WebView`. In this step, we will create a subclass of `WebViewFragment` that adds in a bit of EmPubLite-specific business logic.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Fragment > "Fragment (Blank)" from the context menu. That will bring up a new-fragment dialog:

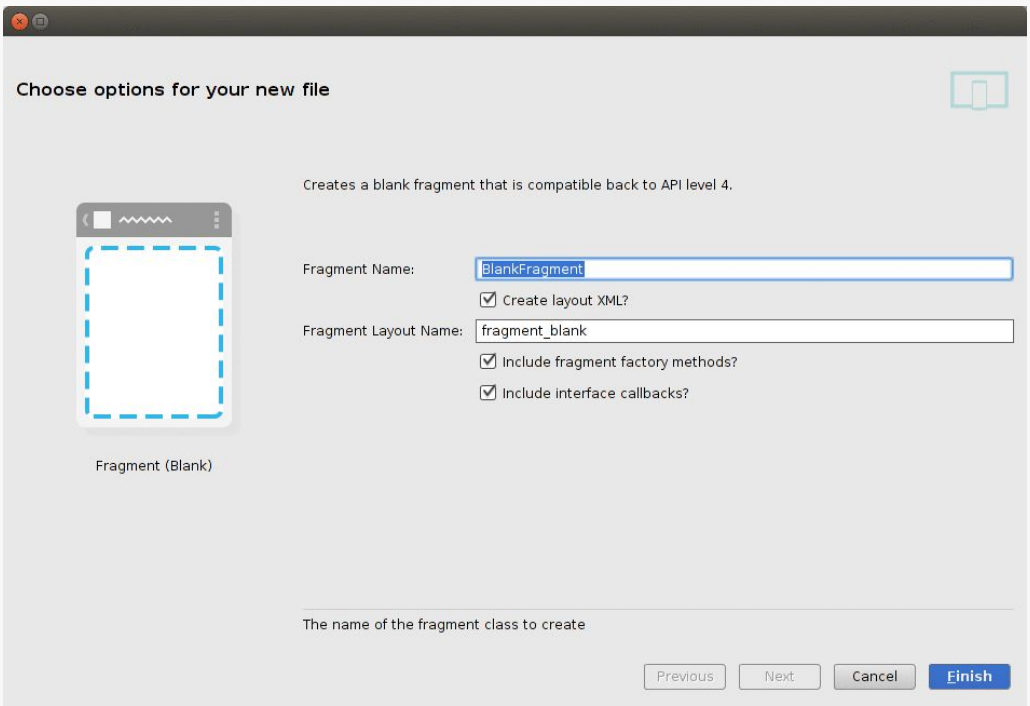


Figure 242: Android Studio New Fragment Dialog

Fill in `SimpleContentFragment` for the "Fragment Name" and uncheck all three checkboxes. Then, click Finish to create the fragment class.

Then, replace the contents of the fragment class with the following code:

```
package com.commonsware.empublite;

import android.annotation.SuppressLint;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.webkit.WebViewFragment;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public class SimpleContentFragment extends WebViewFragment {
  private static final String KEY_FILE = "file";

  protected static SimpleContentFragment newInstance(String file) {
    SimpleContentFragment f = new SimpleContentFragment();
    Bundle args = new Bundle();

    args.putString(KEY_FILE, file);
    f.setArguments(args);

    return (f);
  }

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setRetainInstance(true);
  }

  @SuppressLint("SetJavaScriptEnabled")
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result =
        super.onCreateView(inflater, container, savedInstanceState);

    getWebView().getSettings().setJavaScriptEnabled(true);
    getWebView().getSettings().setSupportZoom(true);
    getWebView().getSettings().setBuiltInZoomControls(true);
    getWebView().loadUrl(getPage());

    return (result);
  }

  private String getPage() {
    return (getArguments().getString(KEY_FILE));
  }
}
```

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

You can also delete the `hello_blank_fragment` string from `res/values/strings.xml`.

Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. Fill in

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`SimpleContentFragment` in the "Name" field. Then, click the "Browse..." button next to the "Superclass" field and find `WebViewFragment` to set as the superclass. Then, click "Finish" on the new-class dialog to create the `SimpleContentFragment` class.

Then, replace its contents with the code shown in the Android Studio instructions above.

Step #2: Examining SimpleContentFragment

`SimpleContentFragment` is simple, with a total of four methods:

- `onCreate()`, where we call `setRetainInstance(true)` — the utility of this will be examined in greater detail in an upcoming chapter.
- `onCreateView()`, where we chain to the superclass (to have it create the `WebView`), then configure it to accept JavaScript and support zoom operations. We then have it load some content, retrieved in the form of a URL from a private `getPage()` method. Finally, we return what the superclass returned from `onCreateView()` — effectively, we are simply splicing in our own configuration logic.
- a `newInstance()` static factory method. This method creates an instance of `SimpleContentFragment`, takes a passed-in `String` (pointing to the file to load), puts it in a `Bundle` identified as `KEY_FILE`, hands the `Bundle` to the fragment as its arguments, and returns the newly-created `SimpleContentFragment`.
- `getPage()`, where it returns a value out of the "arguments" `Bundle` supplied to the fragment — specifically the string identified as `KEY_FILE`.

This means that anyone wanting to use `SimpleContentFragment` should use the factory method, to provide the path to the content to load. We will see why we implemented `SimpleContentFragment` this way in the next chapter.

In Our Next Episode…

... we will set up horizontal swiping of book chapters in our tutorial project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Swiping with ViewPager

Android, over the years, has put increasing emphasis on UI design and having a fluid and consistent user experience (UX). While some mobile operating systems take "the stick" approach to UX (forcing you to abide by certain patterns or be forbidden to distribute your app), Android takes "the carrot" approach, offering widgets and containers that embody particular patterns that they espouse. The action bar, for example, grew out of this and is now the backbone of many Android activities.

Another example is the `ViewPager`, which allows the user to swipe horizontally to move between different portions of your content. However, `ViewPager` is not distributed as part of the firmware, but rather via the Android Support package. Hence, even though `ViewPager` is a relatively new widget, you can use it on Android 1.6 and up.

This chapter will focus on where you should apply a `ViewPager` and how to set one up.

Swiping Design Patterns

In 2012, Google released Android design guidelines as an adjunct to the existing developer documentation. This site outlines many aspects of UI and UX design for Android, from recommended sizing to maintaining platform fidelity instead of mimicking another mobile operating system.

They have a page dedicated to "swipe views", where they outline the scenario for using horizontal swiping: moving from peer to peer in sequence in a collection of content:

- Email messages in a folder or label

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Chapters in an ebook
- Tabs in a collection of tabs

The primary way to implement this pattern in Android is the `ViewPager`.

Pieces of a Pager

`AdapterView` classes, like `ListView`, work with `Adapter` objects, like `ArrayAdapter`. `ViewPager`, however, is not an `AdapterView`, despite adopting many of the patterns from `AdapterView`. `ViewPager`, therefore, does not work with an `Adapter`, but instead with a `PagerAdapter`, which has a slightly different API.

Android ships two `PagerAdapter` implementations in the Android Support package: `FragmentPagerAdapter` and `FragmentStatePagerAdapter`. The former is good for small numbers of fragments, where holding them all in memory at once will work. `FragmentStatePagerAdapter` is for cases where holding all possible fragments to be viewed in the `ViewPager` would be too much, where Android will discard fragments as needed and hold onto the (presumably smaller) states of those fragments instead.

Paging Fragments

The simplest way to use a `ViewPager` is to have it page fragments in and out of the screen based on user swipes.

To see this in action, this section will examine the `ViewPager/Fragments` sample project.

The project has a dependency on the Android Support package, in order to be able to use `ViewPager`. In Android Studio, this is a `compile` statement in the `dependencies` closure of `build.gradle`:

```
dependencies {
  compile 'com.android.support:support-v13:21.0.3'
}
```

In Eclipse, this is a bare JAR in the project's `libs/` directory.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Activity Layout

The layout used by the activity just contains the `ViewPager`. Note that since `ViewPager` is not in the `android.widget` package, we need to fully-qualify the class name in the element:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager xmlns:android="http://schemas.android.com/apk/
res/android"
  android:id="@+id/pager"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
</android.support.v4.view.ViewPager>
```

Note that `ViewPager` is not available for drag-and-drop in the IDE graphical designers, probably because it comes from the Android Support package and therefore is not available to all projects.

The Activity

As you see, the `ViewPagerFragmentDemoActivity` itself is blissfully small:

```
package com.commonsware.android.pager;

import android.app.Activity;
import android.os.Bundle;
import android.support.v4.view.ViewPager;

public class ViewPagerFragmentDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ViewPager pager=(ViewPager)findViewById(R.id.pager);

    pager.setAdapter(new SampleAdapter(getFragmentManager()));
  }
}
```

All we do is load the layout, retrieve the `ViewPager` via `findViewById()`, and provide a `SampleAdapter` to the `ViewPager` via `setAdapter()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The PagerAdapter

Our `SampleAdapter` inherits from `FragmentPagerAdapter` and implements two required callback methods:

- `getCount()`, to indicate how many pages will be in the `ViewPager`, and
- `getItem()`, which returns a `Fragment` for a particular position within the `ViewPager` (akin to `getView()` in a classic `Adapter`)

```
package com.commonsware.android.pager;

import android.app.Fragment;
import android.app.FragmentManager;
import android.support.v13.app.FragmentPagerAdapter;

public class SampleAdapter extends FragmentPagerAdapter {
  public SampleAdapter(FragmentManager mgr) {
    super(mgr);
  }

  @Override
  public int getCount() {
    return(10);
  }

  @Override
  public Fragment getItem(int position) {
    return(EditorFragment.newInstance(position));
  }
}
```

Here, we say that there will be 10 pages total, each of which will be an instance of an `EditorFragment`. In this case, rather than use the constructor for `EditorFragment`, we are using a `newInstance()` factory method. The rationale for that will be explained in the next section.

The Fragment

`EditorFragment` will host a full-screen `EditText` widget, for the user to enter in a chunk of prose, as is defined in the `res/layout/editor.xml` resource:

```
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/editor"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:inputType="textMultiLine"
  android:gravity="left|top"
  />
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We want to pass the position number of the fragment within the `ViewPager`, simply to customize the hint displayed in the `EditText` before the user types in anything. With normal Java objects, you might pass this in via the constructor, but it is not a good idea to implement a constructor on a `Fragment`. Instead, the recipe is to create a static factory method (typically named `newInstance()`) that will create the `Fragment` and provide the parameters to it by updating the fragment's "arguments" (a `Bundle`):

```
  static EditorFragment newInstance(int position) {
    EditorFragment frag=new EditorFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_POSITION, position);
    frag.setArguments(args);

    return(frag);
  }
```

You might be wondering why we are bothering with this `Bundle`, instead of just using a regular data member. The arguments `Bundle` is part of our "saved instance state", for dealing with things like screen rotations — a concept we will get into later in the book. For the moment, take it on faith that this is a good idea.

In `onCreateView()` we inflate our `R.layout.editor` resource, get the `EditText` from it, get our position from our arguments, format a hint containing the position (using a string resource), and setting the hint on the `EditText`:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.editor, container, false);
    EditText editor=(EditText)result.findViewById(R.id.editor);
    int position=getArguments().getInt(KEY_POSITION, -1);

    editor.setHint(String.format(getString(R.string.hint), position + 1));

    return(result);
  }
```

The Result

When initially launched, the application shows the first fragment:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 243: ViewPager on Android 4.3, Showing First Editor

However, you can horizontally swipe to get to the next fragment:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 244: A ViewPager in Use on Android 4.0.3

Swiping works in both directions, so long as there is another page in your desired direction.

Paging Other Stuff

You do not have to use fragments inside a `ViewPager`. A regular `PagerAdapter` actually hands `View` objects to the `ViewPager`. The supplied fragment-based `PagerAdapter` implementations get the `View` from a fragment and use that, but you are welcome to create your own `PagerAdapter` that eschews fragments.

Hence, if you want `ViewPager` to page things other than fragments, the solution is to not use `FragmentPagerAdapter` or `FragmentStatePagerAdapter`, but instead create your own implementation of the `PagerAdapter` interface, one that avoids the use of fragments.

We will see an example of this in a later chapter, where we also examine how to have more than one page of the `ViewPager` be visible at a time.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Indicators

By itself, there is no visual indicator of where the user is within the set of pages contained in the `ViewPager`. In many instances, this will be perfectly fine, as the pages themselves will contain cues as to position. However, even in those cases, it may not be completely obvious to the user how many pages there are, which directions for swiping are active, etc.

Hence, you may wish to attach some other widget to the `ViewPager` that can help clue the user into where they are within "page space".

PagerTitleStrip and PagerTabStrip

The primary built-in indicator options available to use are `PagerTitleStrip` and `PagerTabStrip`. As the name suggests, `PagerTitleStrip` is a strip that shows titles of your pages. `PagerTabStrip` is much the same, but the titles are formatted somewhat like tabs, and they are clickable (switching you to the clicked-upon page), whereas `PagerTitleStrip` is non-interactive.

To use either of these, you first must add it to your layout, inside your `ViewPager`, as shown in the `res/layout/main.xml` resource of the ViewPager/Indicator sample project, a clone of the `ViewPager/Fragments` project that adds a `PagerTabStrip` to our UI:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager xmlns:android="http://schemas.android.com/apk/
res/android"
  android:id="@+id/pager"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <android.support.v4.view.PagerTabStrip
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="top"/>

</android.support.v4.view.ViewPager>
```

Here, we set the `android:layout_gravity` of the `PagerTabStrip` to `top`, so it appears above the pages. You could similarly set it to `bottom` to have it appear below the pages.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Our SampleAdapter needs another method: getPageTitle(), which will return the title to display in the PagerTabStrip for a given position:

```
package com.commonsware.android.pager2;

import android.app.Fragment;
import android.app.FragmentManager;
import android.content.Context;
import android.support.v13.app.FragmentPagerAdapter;

public class SampleAdapter extends FragmentPagerAdapter {
  Context ctxt=null;

  public SampleAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);
    this.ctxt=ctxt;
  }

  @Override
  public int getCount() {
    return(10);
  }

  @Override
  public Fragment getItem(int position) {
    return(EditorFragment.newInstance(position));
  }

  @Override
  public String getPageTitle(int position) {
    return(EditorFragment.getTitle(ctxt, position));
  }
}
```

Here, we call a static getTitle() method on EditorFragment. That is a refactored bit of code from our former onCreateView() method, where we create the string for the hint — we will use the hint text as our page title:

```
package com.commonsware.android.pager2;

import android.app.Fragment;
import android.content.Context;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;

public class EditorFragment extends Fragment {
  private static final String KEY_POSITION="position";

  static EditorFragment newInstance(int position) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    EditorFragment frag=new EditorFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_POSITION, position);
    frag.setArguments(args);

    return(frag);
  }

  static String getTitle(Context ctxt, int position) {
    return(String.format(ctxt.getString(R.string.hint), position + 1));
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.editor, container, false);
    EditText editor=(EditText)result.findViewById(R.id.editor);
    int position=getArguments().getInt(KEY_POSITION, -1);

    editor.setHint(getTitle(getActivity(), position));

    return(result);
  }
}
```

Figure 245: ViewPager and PagerTabStrip on Android 4.3, Showing Second Page

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Third-Party Indicators

If you want something else for your indicators, besides a strip of page titles, you might wish to check out the ViewPagerIndicator library. This library contains a series of widgets that serve in the same role as `PagerTitleStrip`, with different looks. We will look at one such indicator, `TabPageIndicator`, later in this book.

Visit the Trails!

There is a chapter on advanced `ViewPager` techniques that may interest you!

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #10 - Rigging Up a ViewPager

A `ViewPager` is a fairly slick way to present a digital book. You can have individual portions of the book be accessed by horizontal swiping, with the prose within a portion accessed by scrolling vertically. While not offering "page-at-a-time" models used by some book reader software, it is **much** simpler to set up.

So, that's the approach we will use with EmPubLite. Which means, among other things, that we need to add a `ViewPager` to the app.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Add a ViewPager to the Layout

Right now, the layout for `EmPubLiteActivity` just has a `ProgressBar`. We need to augment that to have our `ViewPager` as well, set up such that we can show either the `ProgressBar` (while we load the book) or the `ViewPager` as needed.

Since `ViewPager` is not available for drag-and-drop through the IDE graphical layout editors, even IDE users are going to have to dive into the layout XML this time.

Open up `res/layout/main.xml` and switch to the appropriate sub-tab to see the raw XML. As a child of the `<RelativeLayout>`, after the `<ProgressBar>`, add a `<android.support.v4.view.ViewPager>` element as follows:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:visibility="gone"/>
```

This adds the `ViewPager`, also having it fill the parent, but with the visibility initially set to gone, meaning that the user will not see it.

The entire layout should now resemble:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".EmPubLiteActivity">

  <ProgressBar
    android:id="@+id/progressBar1"
    style="?android:attr/progressBarStyleLarge"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"/>

  <android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:visibility="gone"/>

</RelativeLayout>
```

Step #2: Obtaining Our ViewPager

We will be referencing the `ViewPager` from a few places in the activity, so we may as well get a reference to it and hold onto it in a data member, for easy access.

Add a data member to `EmPubLiteActivity`:

```
private ViewPager pager=null;
```

You will also need to add an import for `android.support.v4.view.ViewPager` to get this to compile.

Then, in `onCreate()`, after the call to `setContentView(R.layout.main)`, use `findViewById()` to retrieve the `ViewPager` and store it in the `pager` data member:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
pager=(ViewPager)findViewById(R.id.pager);
```

You may see a warning that `pager` is not used – do not worry, as we will be using it soon enough.

Step #3: Creating a ContentsAdapter

A `ViewPager` needs a `PagerAdapter` to populate its content, much like a `ListView` needs a `ListAdapter`. We cannot completely construct a `PagerAdapter` yet, as we still need to learn how to load up our book content from files. But, we can get part-way towards having a useful `PagerAdapter` now.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `ContentsAdapter` as the name and click OK to create the empty class.

Then, replace the generated `ContentsAdapter.java` file with the following content:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.Fragment;
import android.support.v13.app.FragmentStatePagerAdapter;

public class ContentsAdapter extends FragmentStatePagerAdapter {
  public ContentsAdapter(Activity ctxt) {
    super(ctxt.getFragmentManager());
  }

  @Override
  public Fragment getItem(int arg0) {
    // TODO Auto-generated method stub
    return null;
  }

  @Override
  public int getCount() {
    // TODO Auto-generated method stub
    return 0;
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. Fill in `ContentsAdapter` in the "Name" field. Then, click the "Browse..." button next to the "Superclass" field and find `FragmentStatePagerAdapter - android.support.v13.app` to set as the superclass. Note that there are two `FragmentStatePagerAdapter` implementations, and you want the one with `v13`, not `v4`, in the package name. Then, click "Finish" on the new-class dialog to create the `ContentsAdapter` class.

This will immediately show an error in the Eclipse editor, as `FragmentStatePagerAdapter` requires a public constructor, and we do not have one yet. So, add the following constructor implementation to the class:

```
  public ContentsAdapter(Activity ctxt) {
    super(ctxt.getFragmentManager());
  }
```

This simply chains to the superclass, supplying the requisite `FragmentManager` instance, culled from our parent activity. Note that you will need to add an import for `android.app.Activity` to the class.

Step #4: Setting Up the ViewPager

Let's add a few more lines to the bottom of `onCreate()` of `EmPubLiteActivity`, to set up `ContentsAdapter` and attach it to the `ViewPager`:

```
    adapter=new ContentsAdapter(this);
    pager.setAdapter(adapter);
    findViewById(R.id.progressBar1).setVisibility(View.GONE);
    pager.setVisibility(View.VISIBLE);
```

This will require a new data member:

```
private ContentsAdapter adapter=null;
```

It will also require an import for `android.view.View`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What we are doing is creating our `ContentsAdapter` instance, associating it with the `ViewPager`, and toggling the visibility of the `ProgressBar` (making it `GONE`) and the `ViewPager` (making it `VISIBLE`).

The net effect, if you run this modified version of the app, is that we no longer see the `ProgressBar`. Instead, we have a big blank area, taken up by our empty `ViewPager`:

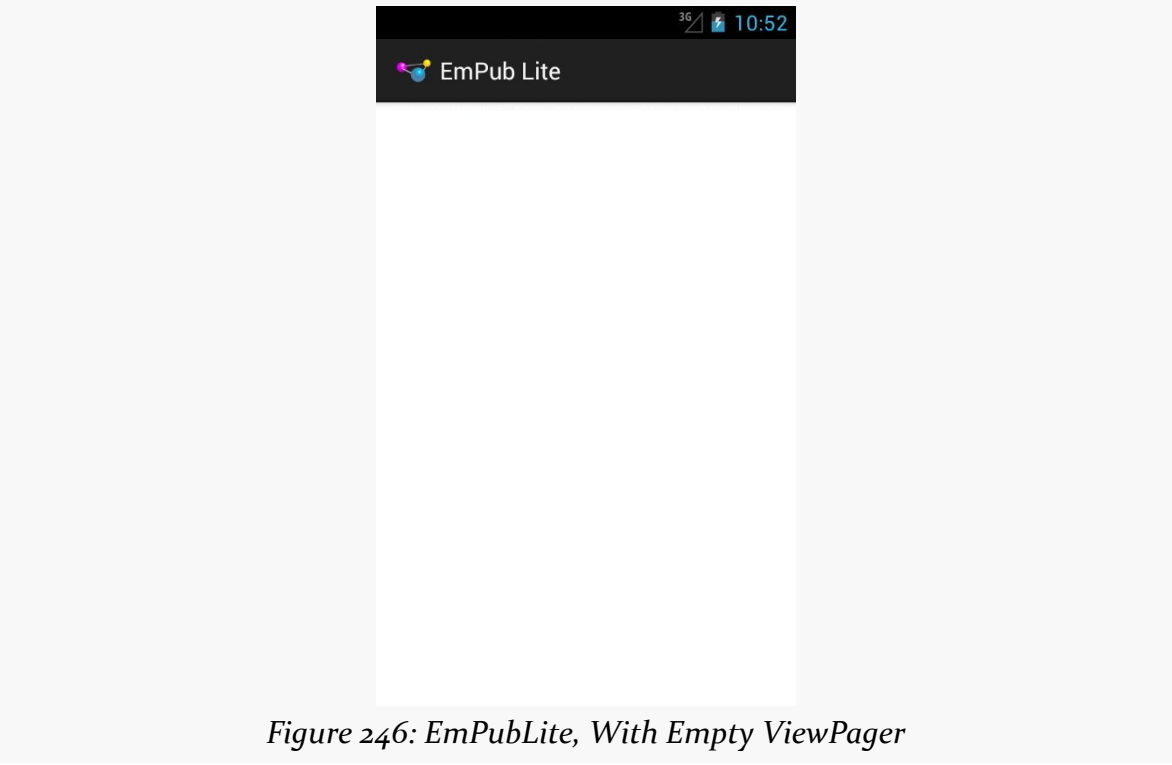


Figure 246: EmPubLite, With Empty ViewPager

The `ViewPager` is empty simply because our `ContentsAdapter` returned `0` from `getCount()`, indicating that there are no pages to be displayed. The `ProgressBar` is no longer visible because we are immediately swapping in the `ViewPager`, but in the future, if it takes some time to get the `ViewPager` set up (e.g., disk I/O), we will see the `ProgressBar` briefly.

In Our Next Episode…

… we will finish our "help" and "about" screens in our tutorial project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Resource Sets and Configurations

Devices sometimes change while users are using them, in ways that our application will care about:

- The user might rotate the screen from portrait to landscape, or vice versa
- The user might put the device in a car or desk dock, or remove it from such a dock
- The user might put the device in a "netbook dock" that adds a full QWERTY keyboard, or remove it from such a dock
- The user might switch to a different language via the Settings application, returning to our running application afterwards
- And so on

In all of these cases, it is likely that we will want to change what resources we use. For example, our layout for a portrait screen may be too tall to use in landscape mode, so we would want to substitute in some other layout.

This chapter will explore how to provide alternative resources for these different scenarios — called "configuration changes" — and will explain what happens to our activities when the user changes the configuration while we are in the foreground.

What's a Configuration? And How Do They Change?

Different pieces of Android hardware can have different capabilities, such as:

- Different screen sizes
- Different screen densities (dots per inch)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Different number and capabilities of cameras
- Different mix of radios (GSM? CDMA? GPS? Bluetooth? WiFi? NFC? something else?)
- And so on

Some of these, in the eyes of the core Android team, might drive the selection of resources, like layouts or drawables. Different screen sizes might drive the choice of layout. Different screen densities might drive the choice of drawable (using a higher-resolution image on a higher-density device). These are considered part of the device's "configuration".

Other differences — ones that do not drive the selection of resources — are not part of the device's configuration but merely are "features" that some devices have and other devices do not. For example, cameras and Bluetooth and WiFi are features.

Some parts of a configuration will only vary based on different devices. A screen will not change density on the fly, for example. But some parts of a configuration can be changed during operation of the device, such as orientation (portrait vs. landscape) or language. When a configuration switches to something else, that is a "configuration change", and Android provides special support for such events to help developers adjust their applications to match the new configuration.

Configurations and Resource Sets

One set of resources may not fit all situations where your application may be used. One obvious area comes with string resources and dealing with internationalization (I18N) and localization (L10N). Putting strings all in one language works fine — probably at least for the developer — but only covers one language.

That is not the only scenario where resources might need to differ, though. Here are others:

1. *Screen orientation*: is the screen in a portrait orientation? Landscape?
2. *Screen size*: is this something sized like a phone? A tablet? A television?
3. *Screen density*: how many dots per inch does the screen have? Will we need a higher-resolution edition of our icon so it does not appear too small?
4. *Keyboard*: what keyboard does the user have (QWERTY, numeric, neither), either now or as an option?
5. *Other input*: does the device have some other form of input, like a directional pad or click-wheel?

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The way Android currently handles this is by having multiple resource directories, with the criteria for each embedded in their names.

Suppose, for example, you want to support strings in both English and Spanish. Normally, for a single-language setup, you would put your strings in a file named `res/values/strings.xml`. To support both English and Spanish, you could create two folders, `res/values-en/` and `res/values-es/`, where the value after the hyphen is the ISO 639–1 two-letter code for the language you want. Your English-language strings would go in `res/values-en/strings.xml` and the Spanish ones in `res/values-es/strings.xml`. Android will choose the proper file based on the user's device settings.

However, the **better** approach is for you to consider some language to be your default, and put those strings in `res/values/strings.xml`. Then, create other resource directories for your translations (e.g., `res/values-es/strings.xml` for Spanish). Android will try to match a specific language set of resources; failing that, it will fall back to the default of `res/values/strings.xml`. This way, if your app winds up on a device with a language that you do not expect, you at least serve up strings in your chosen default language. Otherwise, if there is no such default, you will wind up with a `ResourceNotFoundException`, and your application will crash.

This, therefore, is the bedrock resource set strategy: have a complete set of resources in the default directory (e.g., `res/layout/`), and override those resources in other resource sets tied to specific configurations as needed (e.g., `res/layout-land/`).

Screen Size and Orientation

Perhaps the most important resource set qualifiers that we have not yet seen are the ones related to screen size and orientation. Here, "orientation" refers to how the device is being held: portrait or landscape.

Orientation is fairly easy, as you can just use `-port` or `-land` as resource set qualifiers to restrict resources in a directory to a specific orientation. The convention is to put landscape resources in a `-land` directory (e.g., `res/layout-land/`) and to put portrait resource in the default directory (e.g., `res/layout/`). However, this is merely a convention, and you are welcome to use `-port` if you prefer.

Screen size is a bit more complicated, simply because the available approaches have changed over the years.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Original: Android-Defined Buckets

Way back in the beginning, with Android 1.0, all screen sizes were created equal... mostly because there was only one screen size, and that mostly because there was only one device.

Android 1.5, however, introduced three screen sizes and associated resource set qualifiers, with a fourth (`-xlarge`) added later:

- `-small` for screens at or under 3" in diagonal size
- `-normal` for screens between 3" and 5" in diagonal size
- `-large` for screens between 5" and 10" in diagonal size
- `-xlarge` for screens at or over 10" in diagonal size

As we will see, these resource set qualifiers establish lower bounds for when a directory's worth of resources will be used. So a `res/layout-normal/` directory will not be used for `-small` screens but would be used for `-normal`, `-large`, and `-xlarge` screens.

The Modern: Developer-Defined Buckets

The problem with the classic size buckets is that they were fairly inflexible. What if you think that so-called "phablets", like the Samsung Galaxy Note series, should have layouts more like phones, while larger tablets, such as the 8.9" Kindle Fire HD, should have layouts more like 10" tablets? That was not possible given the fixed buckets.

Android 3.2 gave us more control. We can have our own buckets for screen size, using the somewhat-confusing `-swNNNdp` resource set qualifier. Here, the `NNN` is replaced by you with a value, measured in `dp`, for the *shortest width* of the screen. "Shortest width" basically means the width of the screen when the device is held in portrait mode. Hence, rather than measuring based on diagonal screen size, as with the classic buckets, your custom buckets are based on the linear screen size of the shortest screen side.

For example, suppose that you wish to consider a dividing line between resources to be at the 7" point — 7" and smaller devices would get one set of layouts, while larger devices would get a different set of layouts. 7" tablets usually have a shortest width of around 3.5" to 3.75", given common aspect ratios. Since 1 `dp` is 1/160th of an inch, those shortest widths equate to 560–600 `dp`. Hence, you might set up a `-sw600dp`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

resource set for your larger layouts, and put the smaller layouts in a default resource set.

Mashups: Width and Height Buckets

Using `-swNNNdp` does not address orientation, as the shortest width is the same regardless of whether the device is held in portrait or landscape. Hence, you would need to add `-swNNNdp-land` as a resource set for landscape resources for your chosen dividing line.

An alternative is to use `-wNNNdp` or `-hNNNdp`. These resource set qualifiers work much like `-swNNNdp`, particularly in terms of what `NNN` means. However, whereas `-swNNNdp` refers to the shortest width, `-wNNNdp` refers the *current* width, and `-hNNNdp` refers to the *current* height. Hence, these change with orientation changes.

About That API Level

`-swNNNdp`, `-wNNNdp`, and `-hNNNdp` were added in API Level 13. Hence, older devices will ignore any resource sets with those qualifiers.

In principle, this might seem like a big problem, for those developers still supporting older devices.

In practice, it is less of an issue than you might expect, simply because the *vast* majority of those older devices were phones, not tablets. The only Android 2.x tablets that sold in any significant quantity were three 7" models:

- the original Kindle Fire
- the original Barnes & Noble NOOK series
- the original Samsung Galaxy Tab

Of those, only the Galaxy Tab had the then-Android Market (now the Play Store). Hence, if you are only distributing via the Play Store, you might be in position to simply ignore pre-API Level 13 tablets. Use `-swNNNdp` to create your dividing line for larger devices, and the Galaxy Tab will simply use the layouts for your smaller devices.

If this concerns you, or you are also supporting the Kindle Fire and early NOOKs, you can use layout aliases to minimize code duplication. For example, suppose that you have a `res/layout/main.xml` that you wanted to have different versions for phones and tablets, and you want to use `-swNNNdp` for your dividing line as to where

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the tablet layouts get used, but you also want to have the older tablets, like the Galaxy Tab, use the following recipe:

- Put your tablet-sized layouts in `res/layout/`, but with *different* filenames (e.g., `res/layout/main_to_be_used_for_tablets.xml`)
- In `res/values-swNNNdp/layouts.xml`, for your chosen value of `NNN`, put aliases (via `<item>` elements) for the original names (via the `name` attribute) pointing to the resources you want to use for `-swNNNdp` devices:

```
<resources>
    <item name="main" type="layout">@layout/main_to_be_used_for_tablets</item>
</resources>
```

- In `res/values-large/layouts.xml`, put those same aliases

Now, both older and newer devices, when referencing the same resource name, will get routed to the right layouts for their screen size.

Coping with Complexity

Where things start to get complicated is when you need to use multiple disparate criteria for your resources.

For example, suppose that you have drawable resources that are locale-dependent, such as a stop sign. You might want to have resource sets of drawables tied to language, so you can substitute in different images for different locales. However, you might also want to have those images vary by density, using higher-resolution images on higher-density devices, so the images all come out around the same physical size.

To do that, you would wind up with directories with multiple resource set qualifiers, such as:

- `res/drawable-ldpi/`
- `res/drawable-mdpi/`
- `res/drawable-hdpi/`
- `res/drawable-xhdpi/`
- `res/drawable-en-rUK-ldpi/`
- `res/drawable-en-rUK-mdpi/`
- `res/drawable-en-rUK-hdpi/`
- `res/drawable-en-rUK-xhdpi/`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- And so on

(with the default language being, say, US English, using a US stop sign)

Once you get into these sorts of situations, though, a few rules come into play, such as:

1. The configuration options (e.g., `-en`) have a particular order of precedence, and they must appear in the directory name in that order. The Android documentation outlines the specific order in which these options can appear. For the purposes of this example, screen size is more important than screen orientation, which is more important than screen density, which is more important than whether or not the device has a keyboard.
2. There can only be one value of each configuration option category per directory.
3. Options are case sensitive

For example, you might want to have different layouts based upon screen size and orientation. Since screen size is more important than orientation in the resource system, the screen size would appear in the directory name ahead of the orientation, such as:

- `res/layout-sw600dp-land/`
- `res/layout-sw600dp/`
- `res/layout-land/`
- `res/layout/`

Choosing The Right Resource

Given that you can have N different definitions of a resource, how does Android choose the one to use?

First, Android tosses out ones that are specifically invalid. So, for example, if the language of the device is `-ru`, Android will ignore resource sets that specify other languages (e.g., `-zh`). The exceptions to this are density qualifiers and screen size qualifiers — we will get to those exceptions later.

Then, Android chooses the resource set that has the desired resource and has *the most important distinct qualifier*. Here, by "most important", we mean the one that appears left-most in the directory name, based upon the directory naming rules

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

discussed above. And, by "distinct", we mean where no other resource set has that qualifier.

If there is no specific resource set that matches, Android chooses the default set — the one with no suffixes on the directory name (e.g., `res/layout/`).

With those rules in mind, let's look at some scenarios, to cover the base case plus the aforementioned exceptions.

Scenario #1: Something Simple

Let's suppose that we have a `main.xml` file in:

- `res/layout-land/`
- `res/layout/`

When we call `setContentView(R.layout.main)`, Android will choose the `main.xml` in `res/layout-land/` if the device is in landscape mode. That particular resource set is valid in that case, and it has the most important distinct qualifier (`-land`). If the device is in portrait mode, though, the `res/layout-land/` resource set does not qualify, and so it is tossed out. That leaves us with `res/layout/`, so Android uses that `main.xml` version.

Scenario #2: Disparate Resource Set Categories

It is possible, though bizarre, for you to have a project with `main.xml` in:

- `res/layout-en/`
- `res/layout-land/`
- `res/layout/`

In this case, if the device's locale is set to be English, Android will choose `res/layout-en/`, regardless of the orientation of the device. That is because `-en` is a more important resource set qualifier — "Language and region" appears higher in the "Table 2. Configuration qualifier names" from the Android documentation than does "Screen orientation" (for `-land`). If the device is not set for English, though, Android will toss out that resource set, at which point the decision-making process is the same as in Scenario #1 above.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Scenario #3: Multiple Qualifiers

Now let's envision a project with `main.xml` in:

- `res/layout-en/`
- `res/layout-land-v11/`
- `res/layout/`

You might think that `res/layout-land-v11/` would be the choice, as it is more specific, matching on two resource set qualifiers versus the one or none from the other resource sets.

(in fact, the author of this book thought this was the choice for many years)

In this case, though, language is more important than either screen orientation or Android API level, so the decision-making process is the similar to Scenario #2 above: Android chooses `res/layout-en/` for English-language devices, `res/layout-land-v11/` for landscape API Level 11+ devices, or `res/layout/` for everything else.

Scenario #4: Multiple Qualifiers, Revisited

Let's change the resource mix, so now we have a project with `main.xml` in:

- `res/layout-land-night/`
- `res/layout-land-v11/`
- `res/layout/`

Here, while `-land` is the most important resource set qualifier, it is not distinct — we have more than one resource set with `-land`. Hence, we need to check which is the next-most-important resource set qualifier. In this case, that is `-night`, as night mode is a more important category than is Android API level, and so Android will choose `res/layout-land-night/` if the device is in night mode. Otherwise, it will choose `res/layout-land-v11/` if the device is running API Level 11 or higher. If the device is not in night mode and is not running API Level 11 or higher, Android will go with `res/layout/`.

Scenario #5: Screen Density

Now, let's look at the first exception to the rules: screen density.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android will *always* accept a resource set that contains a screen density, *even if it does not match the density of the device*. If there is an exact density match, of course, Android uses it. Otherwise, it will use what it feels is the next-best match, based upon how far off it is from the device's actual density and whether the other density is higher or lower than the device's actual density.

The reason for this is that for drawable resources, Android will downsample or upsample the image automatically, so the drawable will appear to be the right size, even though you did not provide an image in that specific density.

The catch is two-fold:

1. Android applies this logic to all resources, not just drawables, so even if there is no exact density match on, say, a layout, Android will still choose a resource from another density bucket for the layout
2. As a side-effect of the previous bullet, if you include a density resource set qualifier, Android will ignore any lower-priority resource set qualifiers (unless there are multiple directories with the same density resource set qualifier, in which case the lower-priority qualifiers serve as the "tiebreaker")

So, now let's pretend that our project has `main.xml` in:

- `res/layout-mdpi/`
- `res/layout-nonav/`
- `res/layout/`

Android will choose `res/layout-mdpi/`, even for `-hdpi` devices that do not have a "non-touch navigation method". While `-mdpi` does not match `-hdpi`, Android will still choose `-mdpi`. If we were dealing with drawables resources, Android would upsample the `-mdpi` image.

Scenario #6: Screen Sizes

If you have resource sets tied to screen size, Android will choose the one that is closest to the actual screen size yet smaller than the actual screen size. Resource sets for screen sizes larger than the actual screen size are ignored.

This works for `-swNNNdp`, `-wNNNdp`, and `-hNNNdp` for all devices. On `-large` or `-xlarge` devices, Android applies the same logic for the classic screen size qualifiers (`-small`, `-normal`, `-large`, `-xlarge`). However, Android does *not* apply this logic for `-small` or `-normal` devices — a `-normal` device will not load a `-small` resource.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Now let's pretend that our project has `main.xml` in:

- `res/layout-normal/`
- `res/layout-land/`
- `res/layout/`

Android will choose `res/layout-normal/` if the device is not `-small`. Otherwise, Android will choose `res/layout-land/` if the device is landscape. If all else fails, Android will choose `res/layout/`.

Similarly, if we have:

- `res/layout-w320dp/`
- `res/layout-land/`
- `res/layout/`

Android will choose `res/layout-w320dp/` for devices whose current screen width is 320dp or higher. Otherwise, Android will choose `res/layout-land/` if the device is landscape. If all else fails, Android will choose `res/layout/`.

API-Versioned Resources

As noted previously in this chapter, the `-vNNN` set of suffixes indicate that the resources in that directory are for the stated API level or higher. So, for example, `res/values-v21/` indicates that the resources in that directory should only be used on API Level 21 (Android 5.0) and higher. Devices running older versions of Android will ignore those resources.

This is a particularly important set of suffixes for dealing with major Android version changes. The look and feel of a stock Android app changed significantly at API Level 11 (Android 3.0) and API Level 21 (Android 5.0). You may find that you want to have different resources starting at those API level split points, so that your UI looks appropriately on all versions of Android that you are supporting.

Use Case: Themes by API Level

One big use case for this feature is having different themes by API level.

Even if your `minSdkVersion` is 11 or higher, you may want to have two different themes for your app:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- One, used from API Level 11–20, based on `Theme.Holo`
- Another, used from API Level 21 onwards, based on `Theme.Material`

Your rough alternative is to use the `appcompat-v7` backport of the action bar and bits of the Material Design aesthetic. For highly stylized apps, or in cases where you are sure that you want Material Design on pre-Android 5.0 devices, `appcompat-v7` is worth considering. But if you want to blend in better on each major native UI variant, you will want to support `Theme.Holo` on Android 3.x and 4.x and `Theme.Material` after that.

The hard work here is setting up your themes themselves, such as what was outlined back in the chapter on the action bar. Having them both be available, depending upon device version, is merely a matter of putting the resources into the proper directories.

For example, take a look at the `ActionBar/VersionedColor` sample project. This is a "mashup" of the `HoloColor` and `MaterialColor` sample projects, where the determination of which theme to use is based on API level.

In the `res/values/` directory, we have a `styles.xml` file that is the same as the one in the `HoloColor` example, just with the filename standardizes to `styles.xml`. It uses a custom theme (`Theme.Apptheme`) generated by the Action Bar Style Generator.

There is also a `res/values-v21/` directory, indicating values resources to be used on API Level 21 and higher. It has the theme originally seen in the `MaterialColor` example, where the style resource is renamed to `Theme.Apptheme`, to match the one defined in `res/values/`.

Then, with `<application>` referencing `Theme.Apptheme`, we get the right action bar on the right device.

Here, having the style resources names be the same is important, as we are referencing the name in the `<application>` element in the manifest. To be able to pull in the right one, we need them both to have the same name. However, resources that are referred to by only one of those themes, such as color and drawable resources, could go in a versioned directory or not, as you see fit. They *have* to go in versioned directories and *have* to have the same names if you want multiple editions where the API level chooses which edition to use.

For example, the `Theme.Material`-based theme defined in `res/values-v21/styles.xml` references three color resources. The file for those resources happens to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

also be in `res/values-v21/` (`colors.xml`). However, since we are not looking to replace those colors based on API level, the `colors.xml` file could be placed in `res/values/` and work just as well. And, if we *did* want to have different colors by API level, we would need those colors defined in all relevant resource sets, such as both `res/values/` *and* `res/values-v21/`.

Default Change Behavior

When you call methods in the Android SDK that load a resource (e.g., the aforementioned `setContentView(R.layout.main)`), Android will walk through those resource sets, find the right resource for the given request, and use it.

But what happens if the configuration changes *after* we asked for the resource? For example, what if the user was holding their device in portrait mode, then rotates the screen to landscape? We would want a `-land` version of our layouts, if such versions exist. And, since we already requested the resources, Android has no good way of handing us revised resources on the fly... except by forcing us to re-request those resources.

So, this is what Android does, by default, to our foreground activity, when the configuration changes on the fly.

Destroy and Recreate the Activity

The biggest thing that Android does is destroy and recreate our activity. In other words:

- Android calls `onPause()`, `onStop()`, and `onDestroy()` on our original instance of the activity
- Android creates a brand new instance of the same activity class, using the same `Intent` that was used to create the original instance
- Android calls `onCreate()`, `onStart()`, and `onResume()` of the new activity instance
- The new activity appears on the screen

This may seem... invasive. You might not expect that Android would wipe out a perfectly good activity, just because the user flicked her wrist and rotated the screen of her phone. However, this is the only way Android has that guarantees that we will re-request all our resources.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Rebuild the Fragments

If your activity is using fragments, the new instance of the activity will contain the same fragments that the old instance of the activity does. This includes both static and dynamic fragments.

By default, Android destroys and recreates the fragments, just as it destroys and recreates the activities. However, as we will see, we do have an option to tell Android to retain certain dynamic fragment instances — for those, it will have the new instance use the same fragment instances as were used by the old activity, instead of creating new instances from scratch.

Recreate the Views

Regardless of whether or not Android recreates all of the fragments, it will call `onCreateView()` of all of the fragments (plus call `onDestroyView()` on the original set of fragments). In other words, Android recreates all of the widgets and containers, to pour them into the new activity instance.

Retain Some Widget State

Android will hold onto the "instance state" of some of the widgets we have in our activity and fragments. Mostly, it holds onto obviously user mutable state, such as:

- What has been typed into an `EditText`
- Whether a `CompoundButton`, like a `CheckBox` or `RadioButton`, is checked or not
- Etc.

Android will collect this information from the widgets of the old activity instance, carry that data forward to the new activity instance, and update the new set of widgets to have that same state.

State Saving Scenarios

When the user rotates the screen, or puts the device in a car dock, or changes the language of the device, your process is not terminated. Your foreground activity will be re-created by default — as will your widgets and fragments — but the process sticks around.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, there are plenty of cases when your process will be terminated once you move into the background. That might be done automatically by Android or manually by the user.

Depending on *how* your process is terminated, there may be ways that the user can return to your app and expect that they will return to it just how they left it. For example, suppose the user is in your app, then presses HOME to move your app to the background. Hours pass, and Android terminates your process to free up memory for other apps. Sometime after that, the user brings up the recent-tasks list and taps on your app in that list. From the *user's* perspective, they should be returning to your app in the same state that they left it when they pressed HOME. However, if your process was terminated, by default you lost all that state.

Some of the techniques for dealing with a configuration change — those involving the "saved instance state `Bundle`" — will *also* help you handle the recent-tasks-list scenario. Other of the techniques — such as retaining a fragment — only help with handling configuration changes and will do nothing for you in terms of the recent-tasks-list scenario. The general rule of thumb, therefore, is to use the `Bundle` where you can, and use other techniques (e.g., retained fragments) where the `Bundle` is inappropriate or inadequate. We will see those techniques in the next section.

However, bear in mind that all of this state is designed for transient data, data that the user will not mind if they never see again. For example, suppose the user is in your app, then presses HOME to move your app to the background. Hours pass, and due to the user having busily used their device, *you "fall off" the recent-tasks list*, as that list will not extend indefinitely. In this case, if the user starts up your app again (e.g., via the home screen launcher icon), you will *not* get any state information back for use. Data that the user filled into the old app instance, where that data *must be remembered* and reused in any future run of your app, will need to be persisted yourself, in a database or other type of file.

With all of that in mind, let's examine our options for dealing with the transient state, with an emphasis on configuration changes.

Your Options for Configuration Changes

As noted, a configuration change is fairly invasive on your activity, replacing it outright with all new content (albeit with perhaps some information from the old activity's widgets carried forward into the new activity's widgets).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Hence, you have several possible approaches for handling configuration changes in any given activity.

Do Nothing

The easiest thing to do, of course, is to do nothing at all. If all your state is bound up in stuff Android handles automatically, you do not need to do anything more than the defaults.

For example, the `ViewPager/Fragments` demo from the preceding chapter works correctly "out of the box". All of our "state" is tied up in `EditText` widgets, which Android handles automatically. So, we can type in stuff in a bunch of those widgets, rotate the screen (e.g., via `<Ctrl>-<F11>` in the emulator on a Windows or Linux PC), and our entered text is retained.

Alas, there are plenty of cases where the built-in behavior is either incomplete or simply incorrect, and we will need to do more work to make sure that our configuration changes are handled properly.

Retain Your Fragments

One approach for handling these sorts of configuration changes is to have Android retain a dynamic fragment.

Here, "retain" means that Android will keep the same fragment instance across the configuration change, detaching it from the original hosting activity and attaching it to a new hosting activity. Since it is the same fragment instance, anything contained inside that instance is itself retained and, therefore, is not lost when the activity is destroyed and recreated.

To see this in action, take a look at the `ConfigChange/Fragments` sample project.

The business logic for this demo (and for all the other demos in this chapter) is that we want to allow the user to pick a contact out of the roster of contacts found on their device or emulator. We will do that by having the user press a "Pick" button, at which time we will display an activity that will let the user pick the contact and return the result to us. Then, we will enable a "View" button, and let the user view the details of the selected contact. The key is that our selected contact needs to be retained across configuration changes — otherwise, the user will rotate the screen, and the activity will appear to forget about the chosen contact.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The activity itself just loads the dynamic fragment, following the recipe seen previously in this book:

```
package com.commonsware.android.rotation.frag;

import android.app.Activity;
import android.os.Bundle;

public class RotationFragmentDemo extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content)==null) {
      getFragmentManager().beginTransaction()
                                  .add(android.R.id.content,
                                       new RotationFragment()).commit();
    }
  }
}
```

The reason for checking for the fragment's existence should now be clearer. Since Android will automatically recreate (or retain) our fragments across configuration changes, we do not want to create a *second* copy of the same fragment when we already have an existing copy.

The fragment is going to use an `R.layout.main` layout resource, with two implementations. One, in `res/layout-land/`, will be used in landscape:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <Button android:id="@+id/pick"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="@string/pick"
    android:enabled="true"
  />
  <Button android:id="@+id/view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="@string/view"
    android:enabled="false"
  />
</LinearLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The portrait edition, in `res/layout/`, is identical save for the orientation of the `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <Button android:id="@+id/pick"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="@string/pick"
    android:enabled="true"
  />
  <Button android:id="@+id/view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="@string/view"
    android:enabled="false"
  />
</LinearLayout>
```

Here is the complete implementation of `RotationFragment`:

```
package com.commonsware.android.rotation.frag;

import android.app.Activity;
import android.app.Fragment;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.provider.ContactsContract;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class RotationFragment extends Fragment implements
    View.OnClickListener {
  static final int PICK_REQUEST=1337;
  Uri contact=null;

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    setRetainInstance(true);

    View result=inflater.inflate(R.layout.main, parent, false);

    result.findViewById(R.id.pick).setOnClickListener(this);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    View v=result.findViewById(R.id.view);

    v.setOnClickListener(this);
    v.setEnabled(contact != null);

    return(result);
  }

  @Override
  public void onActivityResult(int requestCode, int resultCode,
                               Intent data) {
    if (requestCode == PICK_REQUEST) {
      if (resultCode == Activity.RESULT_OK) {
        contact=data.getData();
        getView().findViewById(R.id.view).setEnabled(true);
      }
    }
  }

  @Override
  public void onClick(View v) {
    if (v.getId() == R.id.pick) {
      pickContact(v);
    }
    else {
      viewContact(v);
    }
  }

  public void pickContact(View v) {
    Intent i=
        new Intent(Intent.ACTION_PICK,
                   ContactsContract.Contacts.CONTENT_URI);

    startActivityForResult(i, PICK_REQUEST);
  }

  public void viewContact(View v) {
    startActivity(new Intent(Intent.ACTION_VIEW, contact));
  }
}
```

In `onCreateView()`, we hook up the "Pick" button to a `pickContact()` method. There, we call `startActivityForResult()` with an `ACTION_PICK` `Intent`, indicating that we want to pick something from the `ContactsContract.Contacts.CONTENT_URI` collection of contacts. We will discuss `ContactsContract` in greater detail later in this book. For the moment, take it on faith that Android has such an `ACTION_PICK` activity, one that will display to the user the list of available contacts:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

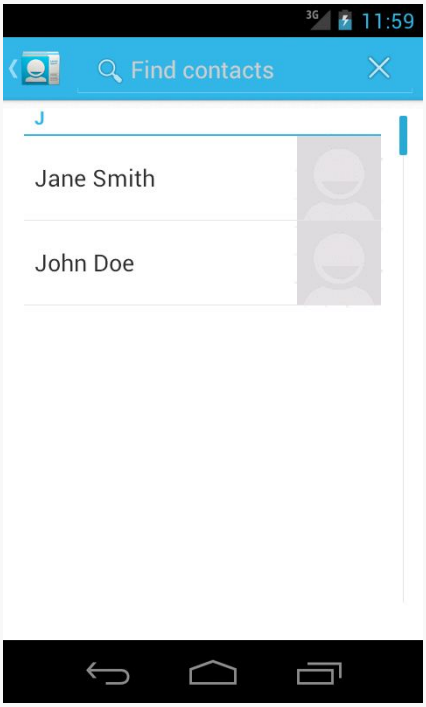


Figure 247: ACTION_PICK of a Contact

If the user picks a contact, control returns to our activity, with a call to `onActivityResult()`. `onActivityResult()` is passed:

- the unique ID we supplied to `startActivityForResult()`, to help identify this result from any others we might be receiving
- `RESULT_OK` if the user did pick a contact, or `RESULT_CANCELED` if the user abandoned the pick activity
- an `Intent` containing the result from the pick activity, which, in this case, will contain a `Uri` representing the selected contact, retrieved via `getData()`

We store that `Uri` in a data member, plus we enable the "View" button, which, when clicked, will bring up an `ACTION_VIEW` activity on the selected contact via its `Uri`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

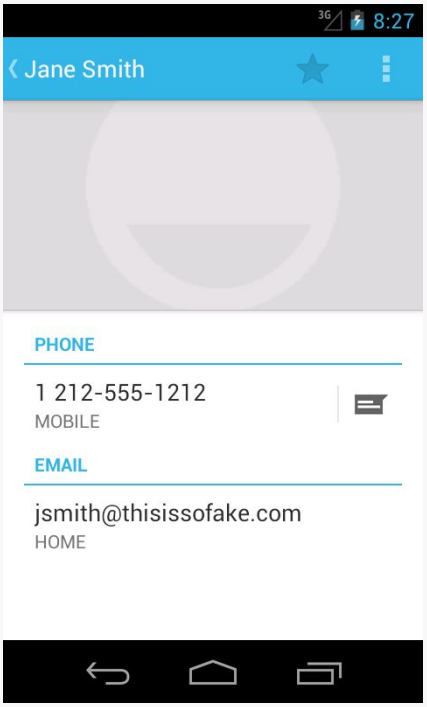


Figure 248: ACTION_VIEW of a Contact

Up in `onCreateView()`, we called `setRetainInstance(true)`. This tells Android to keep this fragment instance across configuration changes. Hence, we can pick a contact in portrait mode, then rotate the screen (e.g., `<Ctrl>-<F11>` in the emulator on Windows or Linux), and view the contact in landscape mode. Even though the activity and the buttons were replaced as a result of the rotation, the fragment was not, and the fragment held onto the `Uri` of the selected contact.

Note that `setRetainInstance()` only works with dynamic fragments, not static fragments. Static fragments are always recreated when the activity is itself destroyed and recreated.

The benefit of this technique, over others, is that you can retain any sort of data you want: any data type, any size, etc. However, this approach does not save state that will be given back to you after your process had been terminated, such as when the user goes back to your app via the recent-tasks list.

Model Fragment

A variation on this theme is the "model fragment". While fragments normally are focused on supplying portions of the UI to a user, that is not really a requirement. A

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

model fragment is one that simply uses `setRetainInstance(true)` to ensure that it sticks around as configurations change. This fragment then holds onto any model data that its host activity needs, so as that activity gets destroyed and recreated, the model data sticks around in the model fragment.

This is particularly useful for data that might not otherwise have a fragment home. For example, imagine an activity whose UI consists entirely of a `ViewPager`, (like the tutorial app). Even though that `ViewPager` might hold fragments, there will be many pages in most pagers. It may be simpler to add a separate, UI-less model fragment and have it hold the activity's data model for the `ViewPager`. This allows the activity to still be destroyed and recreated, and even allows the `ViewPager` to be destroyed and recreated, while still retaining the already-loaded data.

Google recommends using a model fragment instead of using `setRetainInstance(true)` with a regular fragment. The less the retained fragment holds, the less likely it is that you will hold something that you should not be holding, such as a string that needs to be reloaded from a string resource due to a possible locale change. That being said, if you are careful and make sure that all your data members are accounted for properly, using `setRetainInstance(true)` from any fragment can be made safe.

Add to the Bundle

If you want state to be maintained not only for configuration changes, but also for process terminations, you will want to use `onSaveInstanceState()` and `onRestoreInstanceState()`.

You can override `onSaveInstanceState()` in your activity. It is passed a `Bundle`, into which you can store data that should be maintained across the configuration change. The catch is that while `Bundle` looks a bit like it is a `HashMap`, it actually cannot hold arbitrary data types, which limits the sort of information you can retain via `onSaveInstanceState()`. `onSaveInstanceState()` is called around the time of `onPause()` and `onStop()`.

The widget state maintained automatically by Android is via the built-in implementation of `onSaveInstanceState()`. If you override it yourself, typically you will want to chain to the superclass to get this inherited behavior, in addition to putting things into the `Bundle` yourself.

That `Bundle` is passed back to you in two places:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- onCreate()
- onRestoreInstanceState()

Since onCreate() is called in many cases other than due to a configuration change, frequently the passed-in Bundle is null. onRestoreInstanceState(), on the other hand, is only called when there is a Bundle to be used.

To see how this works, take a look at the ConfigChange/Bundle sample project.

Here, RotationBundleDemo is an activity with all the same core business logic as was in our fragment in the preceding demo. Since the activity will be destroyed and recreated on a configuration change, we override onSaveInstanceState() and onRestoreInstanceState() to retain our contact, if one was selected prior to the configuration change:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  if (contact != null) {
    outState.putParcelable("contact", contact);
  }
}

@Override
protected void onRestoreInstanceState(Bundle state) {
  super.onRestoreInstanceState(state);

  contact=state.getParcelable("contact");
  viewButton.setEnabled(contact != null);
}
```

Here, we use putParcelable() to put the Uri into the Bundle. Parcelable is an interface that you can implement on a Java class to allow instance of it to be put into a Bundle. Uri happens to implement Parcelable.

The downside of this approach is that not everything can go into a Bundle. A Bundle cannot hold arbitrary data types, so you cannot put a Socket into a Bundle, for example. Also, this Bundle needs to be fairly small, as it is passed across process boundaries, so you cannot put large objects (e.g., bitmaps) into the Bundle. For those cases, you will have to settle for the retained-fragment approach.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Fragments *and* a Bundle

Fragments also have an `onSaveInstanceState()` method that they can override. It works just like the `Activity` equivalent — you can store data in the supplied `Bundle` that will be supplied back to you later on. The biggest difference is that there is no `onRestoreInstanceState()` method — instead, you are handed the `Bundle` in other lifecycle methods:

- `onCreate()`
- `onCreateView()`
- `onViewCreated()`
- `onActivityCreated()`

We can see this in the `ConfigChange/FragmentBundle` sample project. This is effectively a mashup of the previous two samples: fragments, but using `onSaveInstanceState()` instead of `setRetainInstance(true)`.

Our `RotationFragment` now has an `onSaveInstanceState()` method that looks a lot like the one from the `ConfigChange/Bundle` sample's activity:

```
@Override
public void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  if (contact != null) {
    outState.putParcelable("contact", contact);
  }
}
```

Our `onCreateView()` method examines the passed-in `Bundle`, and if it is not `null` tries to obtain our `contact` from it:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                         Bundle state) {
  View result=inflater.inflate(R.layout.main, parent, false);

  result.findViewById(R.id.pick).setOnClickListener(this);

  View v=result.findViewById(R.id.view);

  v.setOnClickListener(this);

  if (state != null) {
    contact=(Uri)state.getParcelable("contact");
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    v.setEnabled(contact != null);

    return(result);
  }
```

This does not allow our fragment to hold onto arbitrary data, the way `setRetainInstance(true)` does. However, as with `onSaveInstanceState()` at the activity level, there are scenarios that `onSaveInstanceState()` handles that retained fragments will not, such as terminating your process due to low memory, yet the user later uses BACK to return to what should have been your activity (and its fragments).

DIY

In a few cases, even a retained fragment is insufficient, because transferring and re-applying the state would be too complex or too slow. Or, in some cases, the hardware will get in the way, such as when trying to use the `Camera` for taking pictures — a concept we will cover later in this book.

If you are completely desperate, you can tell Android to *not* destroy and recreate the activity on a configuration change... though this has its own set of consequences. To do this:

- Put an `android:configChanges` entry in your `AndroidManifest.xml` file, listing the configuration changes you want to handle yourself versus allowing Android to handle for you
- Implement `onConfigurationChanged()` in your `Activity`, which will be called when one of the configuration changes you listed in `android:configChanges` occurs

Now, for any configuration change you want, you can bypass the whole activity-destruction process and simply get a callback letting you know of the change.

For example, take a look at the ConfigChange/DIY sample project.

In `AndroidManifest.xml`, we add the `android:configChanges` attribute to the `<activity>` element, indicating that we want to handle several configuration changes ourselves:

```
    <activity
      android:name="RotationDIYDemo"
      android:configChanges="keyboardHidden|orientation|screenSize"
      android:label="@string/app_name">
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
```

Many recipes for this will have you handle `orientation` and `keyboardHidden`. However, nowadays, you need to also handle `screenSize` (and, in theory, `smallestScreenSize`), if you have your `android:targetSdkVersion` set to 13 or higher. Note that this will require your build target (e.g., `compileSdkVersion` in Android Studio) to be set to 13 or higher.

Hence, for those particular configuration changes, Android will not destroy and recreate the activity, but instead will call `onConfigurationChanged()`. In the `RotationDIYDemo` implementation, this simply toggles the orientation of the `LinearLayout` to match the orientation of the device:

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
  super.onConfigurationChanged(newConfig);

  LinearLayout container=(LinearLayout)findViewById(R.id.container);

  if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
    container.setOrientation(LinearLayout.HORIZONTAL);
  }
  else {
    container.setOrientation(LinearLayout.VERTICAL);
  }
}
```

Since the activity is not destroyed during a configuration change, we do not need to worry at all about the `Uri` of the selected contact — it is not going anywhere.

The problem with this implementation is twofold:

1. We are not handling all possible configuration changes. If the user, say, puts the device into a car dock, Android will destroy and recreate our activity, and we will lose our selected contact.
2. We might forget some resource that needs to be changed due to a configuration change. For example, if we start translating the strings used by the layouts, and we include `locale` in `android:configChanges`, we not only need to update the `LinearLayout` but also the captions of the `Button` widgets, since Android will not do that for us automatically.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It is these two problems that are why Google does not recommend the use of this technique unless absolutely necessary.

Also, bear in mind that this approach does not help at all for retaining state when your process is terminated and the user returns to your app via the recent-tasks list.

Blocking Rotations

No doubt that you have seen some Android applications that simply ignore any attempt to rotate the screen. Many games work this way, operating purely in landscape mode, regardless of how the device is positioned.

To do this, add `android:screenOrientation="sensorLandscape"`, or possibly `android:screenOrientation="sensorPortrait"`, to your manifest. The "sensor" portions of those names indicate that your app can work in regular or "reverse" versions of the orientation (e.g., "regular" landscape is the device rotated 90 degrees counter-clockwise from portrait, while "reverse" landscape is the device rotated 90 degrees clockwise from portrait). On API Level 18+, you could use `userLandscape` or `userPortrait` instead, as those will honor the user's system-level choice of whether to lock screen rotation or not, defaulting to the behavior of `sensorLandscape` or `sensorPortrait` if the user has not locked screen rotation.

Ideally, you choose landscape, as some devices (e.g., Android TV) can only *be* landscape.

Also note that Android *still* treats this as a configuration change, despite the fact that there is no visible change to the user. Hence, you still need to use one of the aforementioned techniques to handle this configuration change, along with any others (e.g., dock events, locale changes).

And Now, a Word From the Android Project View

Earlier in the book, when introducing Android Studio, we saw the Android project view.

One of the reasons why the Android project view was created was to help you manage resources, particularly across various resource sets.

For example, here is a screenshot of the same Android project, but this time with the `values` resources expanded in the tree:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

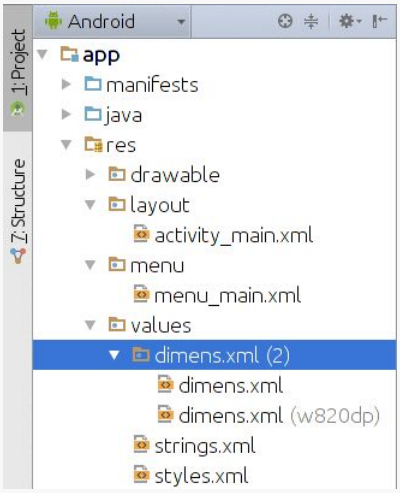


Figure 249: Android Project View, Showing Dimension Resources

What, from a tree standpoint, looks like `res/values/dimens.xml` in turn has two "children", one with just the bare `dimens.xml` filename and one with a "(w820dp)" appended.

This reflects the fact that there are two versions of `dimens.xml`: one in `res/values/` and one in `res/values-w820dp/`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

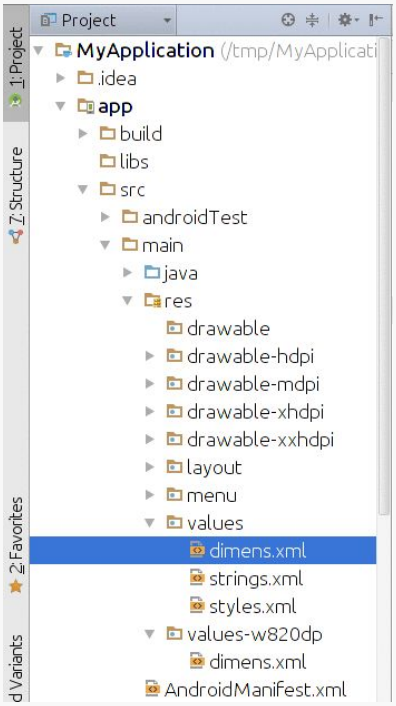


Figure 250: Classic Project View, Showing Dimension Resources

In the Android project view, resources are organized by resource, not by resource set. This can be useful for finding all files that need to be adjusted when you go to adjust one version of the resource, for example.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Material Design Basics

We have already been exposed to `Theme.Material` as part of this book, such as with the action bar.

Android 5.0+, combined with `Theme.Material`, give you a lot of capabilities tied to Google's Material Design aesthetic. In this chapter, we will cover some basic Material Design capabilities that will affect your `Theme.Material` app on Android 5.0+, starting with color.

Your App, in Technicolor!

Some developers want to change the colors used by their app to match some specific color or color palette. In some cases, the colors in question are tied to the app's branding. In other cases, the developer simply wants something different than the stock colors you get from something like `Theme.Holo` or `Theme.Holo.Light`.

Creating custom themes to apply colors to `Theme.Holo` and kin was enough of a pain that a separate theme generator was created for it, independent of the generator for custom action bar colors.

Affecting color changes in your `Theme.Material`-based Android app is vastly simplified — both for the action bar and the widgets — courtesy of `Theme.Material`'s tinting options.

Basic Tinting Options

In the chapter on the action bar, we saw how to set up a custom theme based on `Theme.Material` that had custom color tinting rules that affected the action bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="AppTheme" parent="android:Theme.Material">
    <item name="android:colorPrimary">@color/primary</item>
    <item name="android:colorPrimaryDark">@color/primary_dark</item>
    <item name="android:colorAccent">@color/accent</item>
  </style>
</resources>
```

At that time, we focused on the effects that these tints had on the action bar itself. However, with `Theme.Material`, not only do the tints affect the action bar, but they affect the widgets themselves.

The `BasicMaterial` directory contains clones of some of the basic widget samples outlined earlier in this book, where each includes the custom theme demonstrated for the action bar.

In some cases, the custom tints are not normally visible, such as with a button:

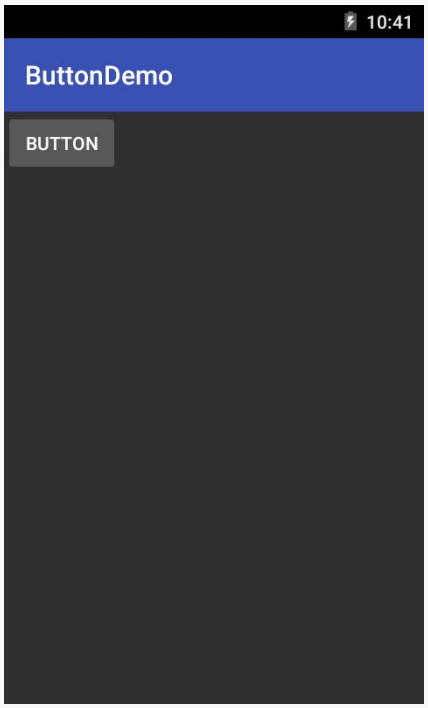


Figure 251: Custom Material Theme for a Button

However, when you tap the button, the animated "ripple" effect shown on the button will use your accent color.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In other cases, the accent color will show up in a more "steady state", such as in a checked `CheckBox`:

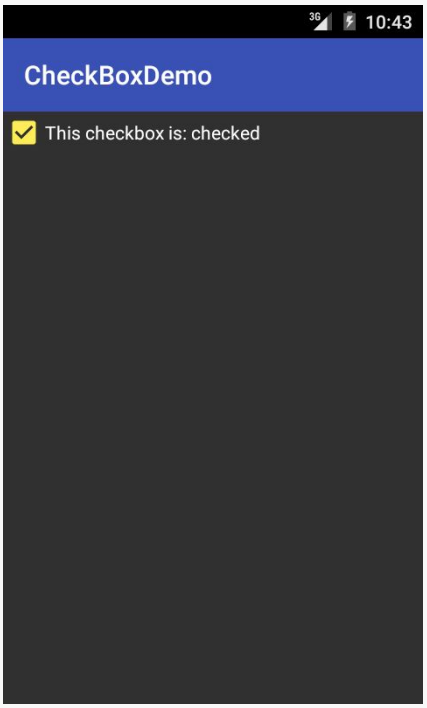


Figure 252: Custom Material Theme for a CheckBox

Similarly, your accent color shows up in things like:

- the "underbar" and drag-cursor in an `EditText`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

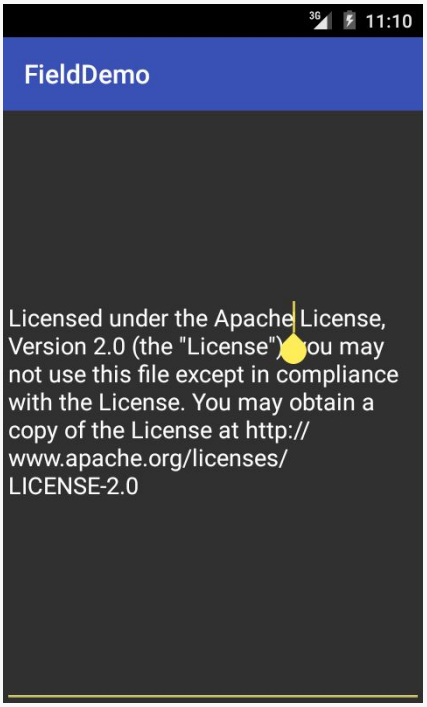


Figure 253: Custom Material Theme for an EditText

- the checked state (and ripple effect when toggling the state) of a `RadioButton`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

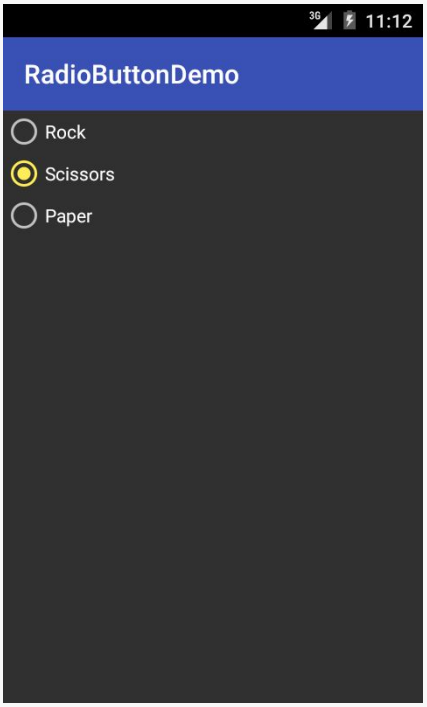


Figure 254: Custom Material Theme for a RadioButton

- the "checked" state of a `Switch`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

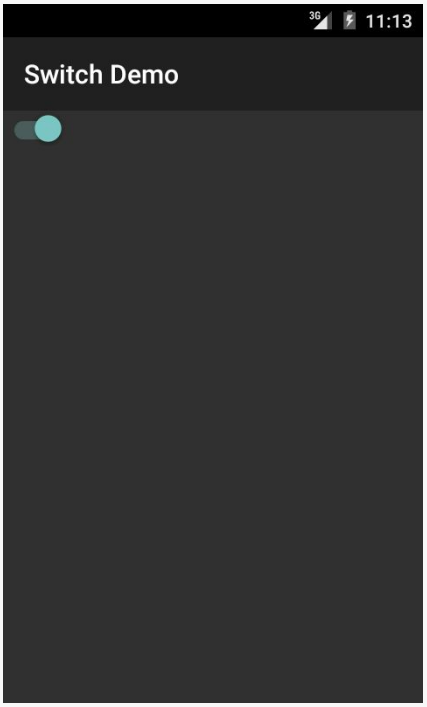


Figure 255: Custom Material Theme for a Switch

Similar effects can be seen in other widgets as well. Hence, your choice of tint values in your custom theme will affect the look of the widgets as well as of the action bar.

Official Google-Approved Colors

Of course, you are welcome to pick whatever colors you like for your theme.

Google has its opinion of what it thinks are good ideas.

As part of the Material Design documentation, you will find a "Color palette" page that outlines possible colors to use.

A Redditor has also published an Android color resource file that contains all of the colors outlined in the Material Design guide.

There is also the material palette site, which generates an color resource file based upon colors that you select from a large grid of color swatches.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tinting the Status Bar

However, when our action bar is a color other than black, white, or shades in between, there can be a color clash with the black background of the status bar. To combat this, Android 5.0 allows you tint the status bar from your Java code. The general guidance is that if your action bar is using your primary color, the status bar should be tinted to the dark primary color.

To do this, as the first thing in your activity's `onCreate()` after chaining to `super.onCreate()` and before setting your content view, you will need to add a bit of code:

```
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.LOLLIPOP) {
      Window window=getWindow();


window.addFlags(WindowManager.LayoutParams.FLAG_DRAWS_SYSTEM_BAR_BACKGROUNDS);
      window.clearFlags(WindowManager.LayoutParams.FLAG_TRANSLUCENT_STATUS);
      window.setStatusBarColor(getResources().getColor(R.color.primary_dark));
    }

    initAdapter();
  }
```

This code snippet assumes that your desired color is found in a color resource named `R.color.primary_dark`, which may need to be changed if you have the resource named differently. Or, if you really want to hard-code a hex color value here (e.g., `0x1a237e`), you can do that instead of looking up the color via a resource.

When run on hardware, you will see the tinted status bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

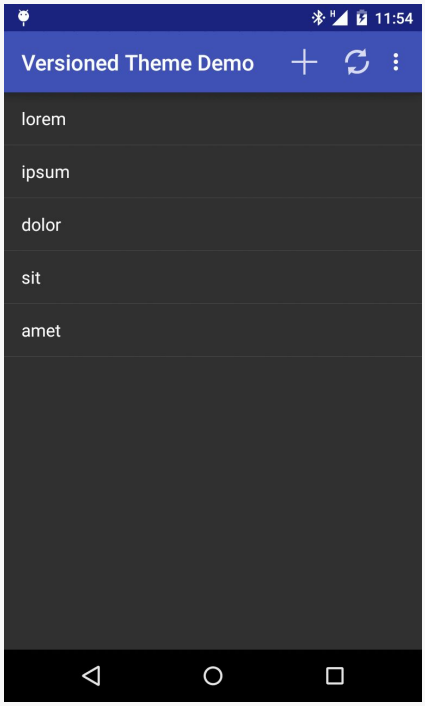


Figure 256: Tinted Status Bar

The Android SDK emulators do not exhibit this behavior at the present time.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Dealing with Threads

Users like snappy applications. Users do not like applications that feel sluggish.

The way to help your application feel snappy is to use the standard threading capabilities built into Android. This chapter will go through the issues involved with thread management in Android and will walk you through some of the options for keeping the user interface crisp and responsive.

The Main Application Thread

When you call `setText()` on a `TextView`, you probably think that the screen is updated with the text you supply, right then and there.

You would be mistaken.

Rather, everything that modifies the widget-based UI goes through a message queue. Calls to `setText()` do not update the screen — they just place a message on a queue telling the operating system to update the screen. The operating system pops these messages off of this queue and does what the messages require.

The queue is processed by one thread, variously called the "main application thread" and the "UI thread". So long as that thread can keep processing messages, the screen will update, user input will be handled, and so on.

However, the main application thread is also used for nearly all callbacks into your activity. Your `onCreate()`, `onClick()`, `onListItemClick()`, and similar methods are all called on the main application thread. While your code is executing in these methods, Android is not processing messages on the queue, and so the screen does not update, user input is not handled, and so on.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This, of course, is bad. So bad, that if you take more than a few seconds to do work on the main application thread, Android may display the dreaded "Application Not Responding" dialog (ANR for short), and your activity may be killed off.

Nowadays, though, the bigger concern is jank.

"Jank", as used in Android, refers to sluggish UI updates, particularly when something is animating. For example, you may have encountered some apps that when you scroll a `ListView` in the app, the `ListView` does not scroll smoothly. Rather, it scrolls jerkily, interleaving periods of rapid movement with periods where the animation is frozen. Most of the time, this is caused by the app's author doing too much work on the main application thread.

Android 4.1 introduced "Project Butter", which, among other things, established a baseline for "doing too much work on the main application thread". We will "drop frames" if we take more than ~16ms per frame (60 frames per second), and dropped frames are the source of jank. Since we may be called many times during a frame, each of our callbacks needs to be very cheap, ideally below 1ms. We will get much more into the issue of jank later in the book, but it is important to understand now that any significant delay in the execution of our code on the main application thread can have visible effects to the user.

Hence, you want to make sure that all of your work on the main application thread happens quickly. This means that anything slow should be done in a background thread, so as not to tie up the main application thread. This includes things like:

1. Internet access, such as sending data to a Web service or downloading an image
2. Significant file operations, since flash storage can be remarkably slow at times
3. Any sort of complex calculations

Fortunately, Android supports threads using the standard `Thread` class from Java, plus all of the wrappers and control structures you would expect, such as the `java.util.concurrent` class package.

However, there is one big limitation: you cannot modify the UI from a background thread. You can only modify the UI from the main application thread. If you call `setText()` on a `TextView` from a background thread, your application will crash, with an exception indicating that you are trying to modify the UI from a "non-UI thread" (i.e., a thread other than the main application thread).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This is a pain.

Getting to the Background

Hence, you need to get long-running work moved into background threads, but those threads need to do something to arrange to update the UI using the main application thread.

There are various facilities in Android for helping with this.

Some are high-level frameworks for addressing this issue for major functional areas. One example of this is the `Loader` framework for retrieving information from databases, and we will examine this in a later chapter.

Sometimes, there are asynchronous options built into other Android operations. For example, when we discuss `SharedPreferences` in a later chapter, we will see that we can persist changes to those preferences synchronously or asynchronously.

And, there are a handful of low-level solutions for solving this problem, ones that you can apply for your own custom business logic.

Asyncing Feeling

One popular approach for handling this threading problem is to use `AsyncTask`. With `AsyncTask`, Android will handle all of the chores of coordinating separate work done on a background thread versus on the UI thread. Moreover, Android itself allocates and removes that background thread. And, it maintains a small work queue, further accentuating the "fire and forget" feel to `AsyncTask`.

The Theory

Theodore Levitt is quoted as saying, with respect to marketing: "People don't want to buy a quarter-inch drill, they want a quarter-inch hole". Hardware stores cannot sell holes, so they sell the next-best thing: devices (drills and drill bits) that make creating holes easy.

Similarly, many Android developers who have struggled with background thread management do not want background threads — they want work to be done off the UI thread, to avoid jank. And while Android cannot magically cause work to not

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

consume UI thread time, Android can offer things that make such background operations easier and more transparent. `AsyncTask` is one such example.

To use `AsyncTask`, you must:

1. Create a subclass of `AsyncTask`
2. Override one or more `AsyncTask` methods to accomplish the background work, plus whatever work associated with the task that needs to be done on the UI thread (e.g., update progress)
3. When needed, create an instance of the `AsyncTask` subclass and call `execute()` to have it begin doing its work

What you do *not* have to do is:

1. Create your own background thread
2. Terminate that background thread at an appropriate time
3. Call all sorts of methods to arrange for bits of processing to be done on the UI thread

AsyncTask, Generics, and Varargs

Creating a subclass of `AsyncTask` is not quite as easy as, say, implementing the `Runnable` interface. `AsyncTask` uses generics, and so you need to specify three data types:

1. The type of information that is needed to process the task (e.g., URLs to download)
2. The type of information that is passed within the task to indicate progress
3. The type of information that is passed when the task is completed to the post-task code

What makes this all the more confusing is that the first two data types are actually used as varargs, meaning that an array of these types is used within your `AsyncTask` subclass.

This should become clearer as we work our way towards an example.

The Stages of AsyncTask

There are four methods you can override in `AsyncTask` to accomplish your ends.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The one you must override, for the task class to be useful, is `doInBackground()`. This will be called by `AsyncTask` on a background thread. It can run as long as it needs to in order to accomplish whatever work needs to be done for this specific task. Note, though, that tasks are meant to be finite – using `AsyncTask` for an infinite loop is not recommended.

The `doInBackground()` method will receive, as parameters, a varargs array of the first of the three data types listed above — the data needed to process the task. So, if your task's mission is to download a collection of URLs, `doInBackground()` will receive those URLs to process.

The `doInBackground()` method must return a value of the third data type listed above — the result of the background work.

You may wish to override `onPreExecute()`. This method is called, from the UI thread, before the background thread executes `doInBackground()`. Here, you might initialize a `ProgressBar` or otherwise indicate that background work is commencing.

Also, you may wish to override `onPostExecute()`. This method is called, from the UI thread, after `doInBackground()` completes. It receives, as a parameter, the value returned by `doInBackground()` (e.g., success or failure flag). Here, you might dismiss the `ProgressBar` and make use of the work done in the background, such as updating the contents of a list.

In addition, you may wish to override `onProgressUpdate()`. If `doInBackground()` calls the task's `publishProgress()` method, the object(s) passed to that method are provided to `onProgressUpdate()`, but in the UI thread. That way, `onProgressUpdate()` can alert the user as to the progress that has been made on the background work. The `onProgressUpdate()` method will receive a varargs of the second data type from the above list — the data published by `doInBackground()` via `publishProgress()`.

A Quick Note About Toasts

In the sample app that follows, we use a `Toast` to let the user know some work has been completed.

A `Toast` is a transient message, meaning that it displays and disappears on its own without user interaction. Moreover, it does not take focus away from the currently-active `Activity`, so if the user is busy writing the next Great Programming Guide, they will not have keystrokes be "eaten" by the message.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Since a Toast is transient, you have no way of knowing if the user even notices it. You get no acknowledgment from them, nor does the message stick around for a long time to pester the user. Hence, the Toast is mostly for advisory messages, such as indicating a long-running background task is completed, the battery has dropped to a low-but-not-too-low level, etc.

Making a Toast is fairly easy. The Toast class offers a static makeText() method that accepts a String (or string resource ID) and returns a Toast instance. The makeText() method also needs the Activity (or other Context) plus a duration. The duration is expressed in the form of the LENGTH_SHORT or LENGTH_LONG constants to indicate, on a relative basis, how long the message should remain visible. Once your Toast is configured, call its show() method, and the message will be displayed.

A Sample Task

As mentioned earlier, implementing an AsyncTask is not quite as easy as implementing a Runnable. However, once you get past the generics and varargs, it is not too bad.

To see an AsyncTask in action, this section will examine the Threads/AsyncTask sample project.

The Fragment and its AsyncTask

We have a ListFragment, named AsyncDemoFragment:

```
package com.commonsware.android.async;

import android.app.ListFragment;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Toast;
import java.util.ArrayList;

public class AsyncDemoFragment extends ListFragment {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };
  private ArrayList<String> model=null;
  private ArrayAdapter<String> adapter=null;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
private AddStringTask task=null;

@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setRetainInstance(true);

  if (model == null) {
    model=new ArrayList<String>();
    task=new AddStringTask();
    task.execute();
  }

  adapter=
      new ArrayAdapter<String>(getActivity(),
                               android.R.layout.simple_list_item_1,
                               model);
}

@Override
public void onViewCreated(View v, Bundle savedInstanceState) {
  super.onViewCreated(v, savedInstanceState);

  getListView().setScrollbarFadingEnabled(false);
  setListAdapter(adapter);
}

@Override
synchronized public void onDestroy() {
  if (task != null) {
    task.cancel(false);
  }

  super.onDestroy();
}

synchronized private void clearTask() {
  task=null;
}

class AddStringTask extends AsyncTask<Void, String, Void> {
  @Override
  protected Void doInBackground(Void... unused) {
    for (String item : items) {
      if (isCancelled())
        break;

      publishProgress(item);
      SystemClock.sleep(400);
    }

    return(null);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    @Override
    protected void onProgressUpdate(String... item) {
      if (!isCancelled()) {
        adapter.add(item[0]);
      }
    }

    @Override
    protected void onPostExecute(Void unused) {
      if (!isCancelled()) {
        Toast.makeText(getActivity(), R.string.done, Toast.LENGTH_SHORT)
             .show();
      }

      clearTask();
    }
  }
}
```

This is another variation on the *lorem ipsum* list of words, used frequently throughout this book. This time, rather than simply hand the list of words to an `ArrayAdapter`, we simulate having to work to create these words in the background using `AddStringTask`, our `AsyncTask` implementation.

In `onCreate()`, we call `setRetainInstance(true)`, so Android will retain this fragment across configuration changes, such as a screen rotation. We then examine a `model` data member. If it is `null`, we know that this is the first time our fragment has been used, so we initialize it to be an `ArrayList` of `String` values, plus kick off our `AsyncTask` (the `AddStringTask` inner class, described below), saving the `AddStringTask` in a `task` data member. Then, in `onViewCreated()`, we set up the adapter and attach it to the `ListView`, also preventing the `ListView` scrollbars from fading away as is their norm.

In the declaration of `AddStringTask`, we use the generics to set up the specific types of data we are going to leverage. Specifically:

1. We do not need any configuration information in this case, so our first type is `Void`
2. We want to pass each string "generated" by our background task to `onProgressUpdate()`, so we can add it to our list, so our second type is `String`
3. We do not have any results, strictly speaking (beyond the updates), so our third type is `Void`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The doInBackground() method is invoked in a background thread. Hence, we can take as long as we like. In a production application, we would be, perhaps, iterating over a list of URLs and downloading each. Here, we iterate over our static list of *lorem ipsum* words, call publishProgress() for each, and then sleep 400 milliseconds to simulate real work being done. We also call isCancelled() on each pass, to see if our task has been cancelled, skipping the work if it has so we can clean up this background thread.

Since we elected to have no configuration information, we should not need parameters to doInBackground(). However, the contract with AsyncTask says we need to accept a varargs of the first data type, which is why our method parameter is Void....

Since we elected to have no results, we should not need to return anything. Again, though, the contract with AsyncTask says we have to return an object of the third data type. Since that data type is Void, our returned object is null.

The onProgressUpdate() method is called on the UI thread, and we want to do something to let the user know we are progressing on loading up these strings. In this case, we simply add the string to the ArrayAdapter, so it gets appended to the end of the list. However, we only do this if we have not already been canceled.

The onProgressUpdate() method receives a String... varargs because that is the second data type in our class declaration. Since we are only passing one string per call to publishProgress(), we only need to examine the first entry in the varargs array.

The onPostExecute() method is called on the UI thread, and we want to do something to indicate that the background work is complete. In a real system, there may be some ProgressBar to dismiss or some animation to stop. Here, we simply raise a Toast and set task to null (via a clearTask() method on the fragment), if our task has not been canceled (checked via another call to isCancelled()).

Since we elected to have no results, we should not need any parameters. The contract with AsyncTask says we have to accept a single value of the third data type. Since that data type is Void, our method parameter is Void unused.

To use AddStringTask, we simply create an instance and call execute() on it. That starts the chain of events eventually leading to the background thread doing its work.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If `AddStringTask` required configuration parameters, we would have not used `Void` as our first data type, and the constructor would accept zero or more parameters of the defined type. Those values would eventually be passed to `doInBackground()`.

Our fragment also has an `onDestroy()` method that calls `cancel()` on the `AsyncTask` if it is still outstanding (`task` is not `null`). This work of cancelling the task and checking to see if the task is cancelled exists for two reasons:

1. Efficiency, as we should skip any serious work that is not needed if our task itself is not needed
2. To avoid a crash if we attempt to raise a `Toast` on a destroyed activity, such as the user launching the activity, then pressing BACK before we complete the background work and display the `Toast`

The Activity and the Results

`AsyncDemo` is a `FragmentActivity` with the standard recipe for kicking off an instance of a dynamic fragment:

```
package com.commonsware.android.async;

import android.app.Activity;
import android.os.Bundle;

public class AsyncDemo extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new AsyncDemoFragment()).commit();
    }
  }
}
```

If you build, install, and run this project, you will see the list being populated in "real time" over a few seconds, followed by a `Toast` indicating completion.

Threads and Configuration Changes

One problem with the default destroy-and-create cycle that activities go through on a configuration change comes from background threads. If the activity has started some background work — through an `AsyncTask`, for example – and then the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

activity is destroyed and re-created, somehow the AsyncTask needs to know about this. Otherwise, the AsyncTask might well send updates and final results to the *old* activity, with the new activity none the wiser. In fact, the new activity might start up the background work *again*, wasting resources.

That is why, in the sample above, we are retaining the fragment instance. The fragment instance holds onto its data model (in this case, the ArrayList of Latin words) and knows not to kick off a new AsyncTask just because the configuration changed. Moreover, we retain that data model, so the new ListView created due to the configuration change can work with a new adapter backed by the old data model, so we do not lose our existing set of Latin words.

We also have to be very careful not to try referring to the activity (via getActivity() on the fragment) from our background thread (doInBackground()). Because, suppose that during the middle of the doInBackground() processing, the user rotates the screen. The activity we work with will change on the fly, on the main application thread, independently of the work being done in the background. The activity returned by getActivity() may not be in a useful state for us while this configuration change is going on.

However, it is safe for us to use getActivity() from onPostExecute(), and even from onProgressUpdate(). For those callbacks, either the configuration change has not yet happened, or it has been completed — we will not be in the middle of the change.

Where Not to Use AsyncTask

AsyncTask, particularly in conjunction with a dynamic fragment, is a wonderful solution for most needs for a background thread.

The key word in that sentence is “most”.

AsyncTask manages a thread pool, from which it pulls the threads to be used by task instances. Thread pools assume that they will get their threads back after a reasonable period of time. Hence, AsyncTask is a poor choice when you do not know how long you need the thread (e.g., thread listening on a socket for a chat client, where you need the thread until the user exits the client).

About the AsyncTask Thread Pool

Moreover, the thread pool that AsyncTask manages has varied in size.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In Android 1.5, it was a single thread.

In Android 1.6, it was expanded to support many parallel threads, probably more than you will ever need.

In Android 3.2, it has shrunk back to a single thread, if your `android:targetSdkVersion` is set to 13 or higher. This was to address concerns about:

- Forking too many threads and starving the CPU
- Developers thinking that there is an ordering dependency between forked tasks, when with the parallel execution there is none

If you wish, starting with API Level 11, you can supply your own `Executor` (from the `java.util.concurrent` package) that has whatever thread pool you wish, so you can manage this more yourself. In addition to the serialized, one-at-a-time `Executor`, there is a built-in `Executor` that implements the old thread pool, that you can use rather than rolling your own.

If your `minSdkVersion` is 11 or higher, use `executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR)` if you specifically want to opt into a multi-thread thread pool. If your `minSdkVersion` is below 11, you will still want to do that... but only on API Level 11+ devices, falling back to `execute()` on the older devices. This static utility method handles this for you:

```
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
static public <T> void executeAsyncTask(AsyncTask<T, ?, ?> task,
                                        T... params) {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, params);
  }
  else {
    task.execute(params);
  }
}
```

To use this, call `executeAsyncTask()`, passing in your `AsyncTask` instance and the parameters you would ordinarily have passed to `execute()`.

An explanation of what we are doing here, in terms of the `@TargetApi` annotation and such, will come later in the book.

Also note that the number of threads in the multiple-thread thread pool has *also* changed over the years. Originally, that pool could climb to as many as 128 threads,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

which was far too many. As of Android 4.4, the thread pool will only grow to "the number of CPU cores * 2 + 1", so on a dual-core device, the thread pool will cap at 5 threads. Further tasks will be queued, up to a maximum of 128 queued tasks.

Alternatives to AsyncTask

There are other ways of handling background threads without using `AsyncTask`:

- You can employ a `Handler`, which has a `handleMessage()` method that will process `Message` objects, dispatched from a background thread, on the main application thread
- You can supply a `Runnable` to be executed on the main application thread to `post()` on any `View`, or to `runOnUiThread()` on `Activity`
- You can supply a `Runnable`, plus a delay period in milliseconds, to `postDelayed()` on any `View`, to run the `Runnable` on the main application thread after *at least* that number of millisecond has elapsed

Of these, the `Runnable` options are the easiest to use.

These can also be used to allow the main application thread to postpone work, to be done *later* on the main application thread. For example, you can use `postDelayed()` to set up a lightweight polling "loop" within an activity, without needing the overhead of an extra thread, such as the one created by `Timer` and `TimerTask`. To see how this works, let's take a peek at the `Threads/PostDelayed` sample project.

This project contains a single activity, named `PostDelayedDemo`:

```
package com.commonsware.android.post;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class PostDelayedDemo extends Activity implements Runnable {
  private static final int PERIOD=5000;
  private View root=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    root=findViewById(android.R.id.content);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void onResume() {
    super.onResume();

    run();
  }

  @Override
  public void onPause() {
    root.removeCallbacks(this);

    super.onPause();
  }

  @Override
  public void run() {
    Toast.makeText(PostDelayedDemo.this, "Who-hoo!", Toast.LENGTH_SHORT)
         .show();
    root.postDelayed(this, PERIOD);
  }
}
```

We want to display a `Toast` every five seconds. To do this, in `onCreate()`, we get our hands on the container for an activity's UI, known as `android.R.id.content`, via `findViewById()`. Then, in `onResume()`, we call a `run()` method on our activity, which displays the `Toast` and calls `postDelayed()` to schedule *itself* (as an implementation of `Runnable`) to be run again in `PERIOD` milliseconds. While our activity is in the foreground, the `Toast` will appear every `PERIOD` milliseconds as a result. Once something else comes to the foreground — such as by the user pressing BACK — our `onPause()` method is called, where we call `removeCallbacks()` to "undo" the `postDelayed()` call.

And Now, The Caveats

Background threads, while eminently possible using `AsyncTask` and kin, are not all happiness and warm puppies. Background threads not only add complexity, but they have real-world costs in terms of available memory, CPU, and battery life.

To that end, there is a wide range of scenarios you need to account for with your background thread, including:

1. The possibility that users will interact with your activity's UI while the background thread is chugging along. If the work that the background thread is doing is altered or invalidated by the user input, you will need to communicate this to the background thread. Android includes many classes

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

in the `java.util.concurrent` package that will help you communicate safely with your background thread.
2. The possibility that the process will be terminated while your work is still going on. This is why in many cases, rather than use an `AsyncTask` or a bare `Thread`, you will wind up using a `Service`, such as an `IntentService`. This will be explored in greater detail later in this book.
3. The possibility that your user will get irritated if you chew up a lot of CPU time and battery life without giving any payback. Tactically, this means using `ProgressBar` or other means of letting the user know that something is happening. Strategically, this means you still need to be efficient at what you do — background threads are no panacea for sluggish or pointless code.
4. The possibility that you will encounter an error during background processing. For example, if you are gathering information off the Internet, the device might lose connectivity. Alerting the user of the problem via a `Notification` and shutting down the background thread may be your best option.

Event Buses

Event-driven programming has been around for nearly a quarter-century. Much of Android's UI model is event-driven, where we find out about these events via callbacks (e.g., `onCreate()` for the "start an activity" event) and registered listeners (e.g., `OnClickListener` for when the user taps on a widget).

However, originally, Android did not have a very fine-grained event or message bus implementation that we as developers could use. The `Intent` system works like a message bus, but it is aimed at inter-process communication (IPC) as much as in-process communication, and that comes with some costs.

However, over time, particularly starting in 2012, event buses started to pop up, and these are very useful for organizing communication within your Android application and across threads. Used properly, an event bus can eliminate the need for `AsyncTask` and the other solutions for communicating back to the main application thread, while simultaneously helping you logically decouple independent pieces of your code.

What Is an Event Bus?

Whether you consider it an "event bus" (or "message bus"), the "publisher/subscriber" (or "pub/sub") pattern, or a subset of the "observer" pattern, the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

programming model where components produce events that others consume for is reasonably common in modern software development.

An event bus is designed to decouple the sources of events from the consumers of those events. Or, as one event bus author put it:

> I want an easy, centralized way to notify code that's interested in specific types of events when those events occur without any direct coupling between the code the publishes an event and the code that receives it.

With the traditional Java listener or observer pattern implementation, the component producing an event needs direct access to consumers of that event. Sometimes, that list of consumers is limited to a single consumer, as with many event handlers associated with Android widgets (e.g., just one `OnClickListener`). But this source-holds-the-sinks coding pattern limits flexibility, as it requires explicit registration by consumers with producers of events, and it may not be that easy for the consumer to *reach* the producer. Furthermore, such direct connections are considered to be a relatively strong coupling between those components, and often times our objective is to have looser coupling.

An event bus provides a standard communications channel (or "bus") that event producers and event consumers can hook into. Event producers merely need to hand the event to the bus; the bus will handle directing those events to relevant consumers. This reduces the coupling between the producers and consumers, sometimes even reducing the amount of code needed to source and sink these events.

OK, But Why Are We Bothering With This?

Later on, we are going to have components other than our activities. In particular, we will have services, which are designed to run briefly in the background to perform some operation. Just as communications between activities tends to be loosely coupled, so too are communications between activities and services. An event bus is a great way for the service to let other pieces of the app know that certain work was done (e.g., "the download is complete, so update the UI").

In the short term, we will use an event bus to have a model fragment let the app know that some data was loaded. In the tutorials, "some data" will be the book contents; in the sample app illustrated in this chapter, "some data" will be some Latin words.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Introducing greenrobot's EventBus

The event bus implementation that we will be using in the tutorials is greenrobot's EventBus, an open source implementation based on the Guava project's event bus. With greenrobot's EventBus, it is fairly easy to send a message from one part of your app to another disparate part of your app.

To illustrate its use, take a look at the `EventBus/AsyncDemo` sample project. This is a reworking of a previous example that used an `AsyncTask` to pretend to download our list of Latin words, populating a `ListView` with those words as they arrive. This sample replaces the `AsyncTask` with a model fragment that will keep track of the words and a background thread that will "download" the words. We will use events raised by the model fragment to let the UI fragment know words as they arrive.

Defining Events

With greenrobot's EventBus, the "events" are objects of arbitrary classes that you define. Each different class represents a different type of event, and you can define as many different event classes as you wish. Those classes do not need to inherit from any special base class, or implement some special interface, or have any magic annotations. They are just classes.

You may wish to put data members, constructors, and accessor methods on the event classes, for any data you wish to pass around specific to the event itself. A `SearchEvent`, for example, might include the search query string as part of the event object.

In our case, we have a `WordReadyEvent` that contains the new word:

```
package com.commonsware.android.eventbus;

class WordReadyEvent {
  private String word;

  WordReadyEvent(String word) {
    this.word=word;
  }

  String getWord() {
    return(word);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Posting Events

To post an event, all you need to do is obtain an instance of an `EventBus` – typically via the `getDefault()` method on `EventBus` — and call `post()` on it, passing in the event to be delivered to any interested party within your app.

With that in mind, let's look at the `ModelFragment` that will be loading in our words:

```
package com.commonsware.android.eventbus;

import android.app.Fragment;
import android.os.Bundle;
import android.os.SystemClock;
import java.util.ArrayList;
import de.greenrobot.event.EventBus;

public class ModelFragment extends Fragment {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };
  private ArrayList<String> model=new ArrayList<String>();
  private boolean isStarted=false;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setRetainInstance(true);

    if (!isStarted) {
      isStarted=true;
      new LoadWordsThread().start();
    }
  }

  public ArrayList<String> getModel() {
    return(new ArrayList<String>(model));
  }

  class LoadWordsThread extends Thread {
    @Override
    public void run() {
      for (String item : items) {
        if (!isInterrupted()) {
          model.add(item);
          EventBus.getDefault().post(new WordReadyEvent(item));
          SystemClock.sleep(400);
        }
      }
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }
}
```

This fragment has no UI — it exists solely to manage a data model on behalf of the rest of the hosting activity. Hence, there is no `onCreateView()` or any other UI logic directly in this fragment.

In `onCreate()`, we call `setRetainInstance(true)`, so that if the user rotates the screen or otherwise triggers a configuration change, our model fragment will survive the change and be attached to the new activity instance. Then, if we have not already started the `LoadWordsThread`, we do so. `LoadWordsThread` iterates over our list of words, sleeps for 400ms to simulate doing real work, adds each word to an `ArrayList` of words that it manages... and calls `post()` to raise a `WordReadyEvent` to let something else know that the model has changed.

Receiving Events

To receive posted events, you need to do three things:

1. Call `register()` on the `EventBus` to tell it that you have an object that wants to receive events
2. Call `unregister()` on the `EventBus` to tell it to stop delivering events to a previously-registered object
3. Implement `onEventMainThread()`, or other `onEvent()` method flavors, to indicate the type of event you want to receive (and to actually process those events)

This sample app has an `AsyncDemoFragment` that performs those three steps:

```
package com.commonsware.android.eventbus;

import android.app.Activity;
import android.app.ListFragment;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import java.util.ArrayList;
import de.greenrobot.event.EventBus;

public class AsyncDemoFragment extends ListFragment {
  private ArrayAdapter<String> adapter=null;
  private ArrayList<String> model=null;

  @Override
  public void onViewCreated(View view, Bundle savedInstanceState) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    adapter=
        new ArrayAdapter<String>(getActivity(),
                                 android.R.layout.simple_list_item_1,
                                 model);

    getListView().setScrollbarFadingEnabled(false);
    setListAdapter(adapter);
  }

  @Override
  public void onAttach(Activity activity) {
    super.onAttach(activity);

    EventBus.getDefault().register(this);
  }

  @Override
  public void onDetach() {
    EventBus.getDefault().unregister(this);

    super.onDetach();
  }

  public void onEventMainThread(WordReadyEvent event) {
    adapter.add(event.getWord());
  }

  public void setModel(ArrayList<String> model) {
    this.model=model;
  }
}
```

The fragment starts by overriding `onViewCreated()`, where we create an `ArrayAdapter` and use that to populate the `ListView`.

The `onAttach()` and `onDetach()` methods are where we indicate to the `EventBus` that this fragment object wants to receive relevant posted events. `onAttach()` calls `register()`; `onDetach()` calls `unregister()`.

The `onEventMainThread()` method, via its parameter, indicates that we are interested in `WordReadyEvents` as they are raised. Our `onEventMainThread()` method will be called for each `WordReadyEvent` passed to `post()` on the `EventBus`. As the method name suggests, `onEventMainThread()` is called on the main application thread, so it is safe for us to update our UI. greenrobot's EventBus is responsible for getting this event to the main application thread — note that we are posting the event from the `LoadWordsThread`, which is a background thread.

In `onEventMainThread()`, we get the newly-added word, which we can add to our `ArrayAdapter`. `add()` on `ArrayAdapter` appends the word to the end of the list and

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

informs the attached `ListView` that the data changed, so the `ListView` can redraw itself.

What is not obvious, though, from the code in this class is how we are getting the `model` that we are using in `onViewCreated()`. `AsyncDemoFragment` has its own `ArrayList` of words, set via the `setModel()` method. Our `ArrayAdapter` is wrapped around this model. But the master copy of the words is being held by the `ModelFragment`. If the `ModelFragment` has the model, and the `AsyncDemoFragment` needs the model, how are the two being connected?

The Activity

That is handled by our hosting activity, as it sets up these two fragments:

```
package com.commonsware.android.eventbus;

import android.app.Activity;
import android.os.Bundle;

public class AsyncDemo extends Activity {
  private static final String MODEL_TAG="model";
  private ModelFragment mFrag=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    mFrag=
        (ModelFragment)getFragmentManager().findFragmentByTag(MODEL_TAG);

    if (mFrag == null) {
      mFrag=new ModelFragment();

      getFragmentManager().beginTransaction().add(mFrag, MODEL_TAG)
                          .commit();
    }

    AsyncDemoFragment demo=

(AsyncDemoFragment)getFragmentManager().findFragmentById(android.R.id.content);

    if (demo == null) {
      demo=new AsyncDemoFragment();
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content, demo).commit();
    }

    demo.setModel(mFrag.getModel());
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }
}
```

In `onCreate()`, we first see if we already have an instance of our model fragment, held by the `FragmentManager` under a `MODEL_TAG` tag. If not, we create an instance of the `ModelFragment` and add it to the `FragmentManager`, under that tag, via a `FragmentTransaction`.

We then see if we already have an instance of our `AsyncDemoFragment`. If not, we create one and add it to the `FragmentManager`, pouring its UI into `android.R.id.content`, via another `FragmentTransaction`.

Then, we connect the two, calling `getModel()` on the `ModelFragment` and handing the result to `setModel()` on the `AsyncDemoFragment`.

When our activity is newly launched, neither fragment exists. Both fragments are created, and the `AsyncDemoFragment` gets its model array from the `ModelFragment`. That array is initially empty. As the `ModelFragment` adds elements to the array, it posts the `WordReadyEvent`, which triggers the `AsyncDemoFragment` to tell the `ArrayAdapter` and `ListView` that the model data changed.

If we undergo a configuration change, the `ModelFragment` is retained, but the `AsyncDemoFragment` is not. Hence, the activity will always be creating an `AsyncDemoFragment`. But the model we give to the `AsyncDemoFragment` may already have words in it, and those words will appear immediately when the `ArrayAdapter` is wrapped around the model. If the `LoadWordsThread` is still running, the new `AsyncDemoFragment` will pick up any new `WordReadyEvents` that are raised, triggering it to update the `ListView` as before.

Visit the Trails!

We will cover much more about jank, and how to detect and diagnose it, in a later chapter.

There are many more features in the greenrobot EventBus implementation. We will see some of those, plus other event bus implementations, in a later chapter on event bus alternatives.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Requesting Permissions

In the late 1990's, a wave of viruses spread through the Internet, delivered via email, using contact information culled from Microsoft Outlook. A virus would simply email copies of itself to each of the Outlook contacts that had an email address. This was possible because, at the time, Outlook did not take any steps to protect data from programs using the Outlook API, since that API was designed for ordinary developers, not virus authors.

Nowadays, many applications that hold onto contact data secure that data by requiring that a user explicitly grant rights for other programs to access the contact information. Those rights could be granted on a case-by-case basis or all at once at install time.

Android is no different, in that it requires permissions for applications to read or write contact data. Android's permission system is useful well beyond contact data, and for content providers and services beyond those supplied by the Android framework.

You, as an Android developer, will frequently need to ensure your applications have the appropriate permissions to do what you want to do with other applications' data. This chapter covers this topic.

You may also elect to require permissions for other applications to use your data or services, if you make those available to other Android components. This will be discussed later in this book.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Mother, May I?

Requesting the use of other applications' data or services requires the `uses-permission` element to be added to your `AndroidManifest.xml` file. Your manifest may have zero or more `uses-permission` elements, all as direct children of the root `manifest` element.

The `uses-permission` element takes a single attribute, `android:name`, which is the name of the permission your application requires:

```
<uses-permission android:name="android.permission.ACCESS_LOCATION" />
```

The stock system permissions all begin with `android.permission` and are listed in the Android SDK documentation for `Manifest.permission`. Third-party applications may have their own permissions, which hopefully they have documented for you. Here are some of the permissions we will see in this book:

1. `INTERNET`, if your application wishes to access the Internet through any means, from raw Java sockets through the `WebView` widget
2. `WRITE_EXTERNAL_STORAGE`, for writing data to external storage
3. `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`, for determining where the device is
4. `CALL_PHONE`, to allow the application to place phone calls directly, without user intervention

Permissions are confirmed at the time the application is installed — the user will be prompted to confirm it is OK for your application to do what the permission calls for.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

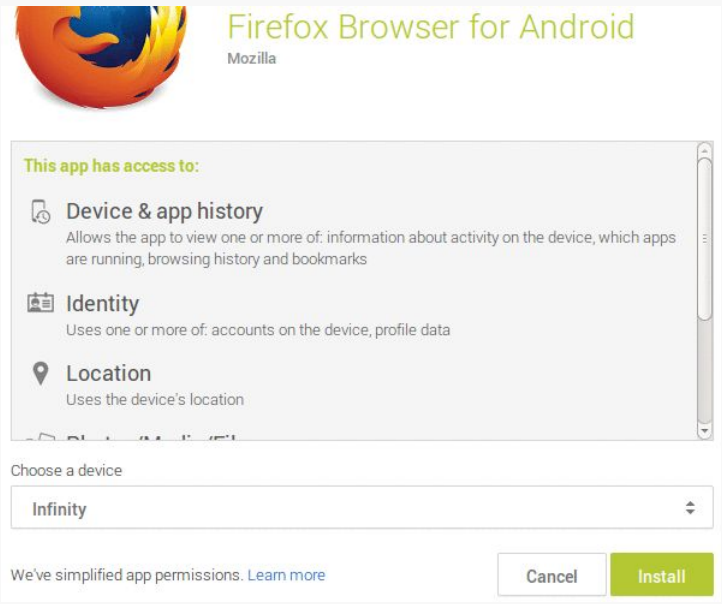


Figure 257: Permission Confirmation Screen, on Play Store Web Site

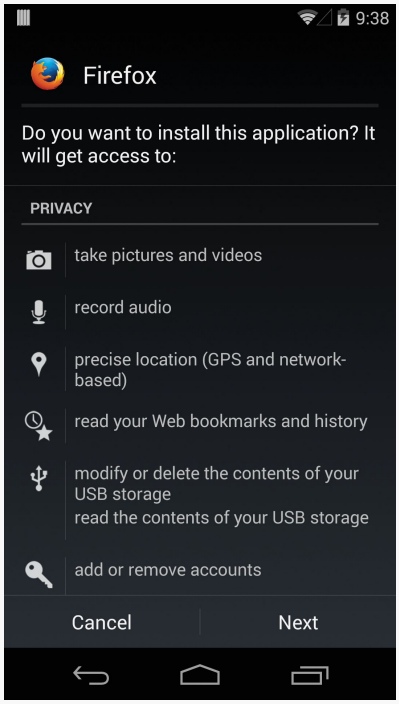


Figure 258: Permission Confirmation Screen, on Android 4.4

Hence, it is important for you to ask for as few permissions as possible and to justify those you ask for, so users do not elect to skip installing your application because

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

you ask for too many unnecessary permissions. Note that users are not asked to confirm permissions when loading an application via USB, such as during development.

If you do not have the desired permission and try to do something that needs it, you should get a `SecurityException` informing you of the missing permission. Note that you will only fail on a permission check if you forgot to ask for the permission — it is impossible for your application to be running and *not* have been granted your requested permissions, at least on conventional versions of Android. So-called "ROM mods" might allow users to revoke already-granted permissions.

New Permissions in Old Applications

Sometimes, Android introduces new permissions that govern behavior that formerly did not require permissions. `WRITE_EXTERNAL_STORAGE` is one example – originally, applications could write to external storage without any permission at all. Android 1.6 introduced `WRITE_EXTERNAL_STORAGE`, required before you can write to external storage. However, applications that were written before Android 1.6 could not possibly request that permission, since it did not exist at the time. Breaking those applications would seem to be a harsh price for progress.

What Android does is "grandfather" in certain permissions for applications supporting earlier SDK versions.

In particular, if you have `<uses-sdk android:minSdkVersion="3">` in your manifest, saying that you support Android 1.5, your application will automatically request `WRITE_EXTERNAL_STORAGE` and `READ_PHONE_STATE`, even if you do not explicitly request those permissions. People installing your application on an Android 1.5 device will see these requests.

Eventually, when you drop support for the older version (e.g., switch to `<uses-sdk android:minSdkVersion="4">`), Android will no longer automatically request those permissions. Hence, if your code really *does* need those permissions, you will need to ask for them yourself.

Permissions: Up Front Or Not At All

The permission system in Android is not especially flexible. Notably, you have to ask for all permissions you might ever need up front, and the user has to agree to all of them or abandon the installation of your app.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This means:

1. You cannot create optional permissions, ones the user could say "no, thanks" to, that your application could react to dynamically
2. You cannot request new permissions after installation, so even if a permission is only needed for some lightly-used feature, you have to ask for it anyway

Hence, it is important as you come up with the feature list for your app that you keep permissions in mind. Every additional permission that you request is a filter that will cost you some portion of your prospective audience. Certain combinations — such as `INTERNET` and `READ_CONTACTS` — will have a stronger effect, as users fear what the combination can do. You will need to decide for yourself if the additional users you will get from having the feature will be worth the cost of requiring the permissions the feature needs to operate.

Signature Permissions

Some permissions listed in the SDK you can request but will not get. These permissions, such as `BRICK`, require your application to be signed by the same signing key as is used to sign the firmware. We will discuss these signing keys and how they work in a later chapter.

Some permissions, like `REBOOT`, require that your application either be signed with the firmware's signing key *or* that your application be "installed on the system partition" (which normally means that the app is distributed pre-installed on the device).

Unfortunately, the Android developer documentation does not tell you the requirements for any given permission. To find out, you will need to examine the platform's `AndroidManifest.xml` file and find your permission in there. For example, here is one edition's definition of the `BRICK` and `REBOOT` permissions:

```
<!-- Required to be able to disable the device (very dangerous!). -->
<permission android:name="android.permission.BRICK"
    android:label="@string/permlab_brick"
    android:description="@string/permdesc_brick"
    android:protectionLevel="signature" />

<!-- Required to be able to reboot the device. -->
<permission android:name="android.permission.REBOOT"
    android:label="@string/permlab_reboot"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:description="@string/permdesc_reboot"
    android:protectionLevel="signature|system" />
```

The `BRICK` permission has an `android:protectionLevel` of `signature`, meaning the app requesting the permission must have the same signing key as does the firmware. Instead, the `REBOOT` permission has `signature|system`, meaning that the app could just be installed on the system partition in order to hold this permission. Here, the pipe operator is a logical `OR`, meaning that the app must either be signed by the firmware's signing key (`signature`) *or* installed on the system partition (`system`).

Requiring Permissions

The XML elements shown from Android's own manifest are `<permission>` elements. These define new permissions to the system.

You can use `<permission>` elements to define your own custom permissions for use with your own apps. This would be important if you are planning on allowing third-party applications to integrate with yours and possibly retrieve data that you are storing. The user probably should "get a vote" on whether that data sharing is allowed. To do that, you could define a permission and declare that one or more of your components (e.g., activities) are protected by that permission. Only third parties that request the permission via `<uses-permission>` will be able to use those components.

We will get into this scenario in greater detail in a later chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Assets, Files, and Data Parsing

Android offers a few structured ways to store data, notably `SharedPreferences` and local SQLite databases. And, of course, you are welcome to store your data "in the cloud" by using an Internet-based service. We will get to all of those topics shortly.

Beyond that, though, Android allows you to work with plain old ordinary files, either ones baked into your app ("assets") or ones on so-called internal or external storage.

To make those files work — and to consume data off of the Internet — you will likely need to employ a parser. Android ships with several choices for XML and JSON parsing, in addition to third-party libraries you can attempt to use.

This chapter focuses on assets, files, and parsers.

Packaging Files with Your App

Let's suppose you have some static data you want to ship with the application, such as a list of words for a spell-checker. Somehow, you need to bundle that data with the application, in a way you can get at it from Java code later on, or possibly in a way you can pass to another component (e.g., `WebView` for bundled HTML files).

There are three main options here: raw resources, XML resources, and assets.

Raw Resources

One way to deploy a file like a spell-check catalog is to put the file in the `res/raw` directory, so it gets put in the Android application `.apk` file as part of the packaging process as a raw resource.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To access this file, you need to get yourself a `Resources` object. From an activity, that is as simple as calling `getResources()`. A `Resources` object offers `openRawResource()` to get an `InputStream` on the file you specify. Rather than a path, `openRawResource()` expects an integer identifier for the file as packaged. This works just like accessing widgets via `findViewById()` – if you put a file named `words.xml` in `res/raw`, the identifier is accessible in Java as `R.raw.words`.

Since you can only get an `InputStream`, you have no means of modifying this file. Hence, it is really only useful for static reference data. Moreover, since it is unchanging until the user installs an updated version of your application package, either the reference data has to be valid for the foreseeable future, or you will need to provide some means of updating the data. The simplest way to handle that is to use the reference data to bootstrap some other modifiable form of storage (e.g., a database), but this makes for two copies of the data in storage. An alternative is to keep the reference data as-is but keep modifications in a file or database, and merge them together when you need a complete picture of the information. For example, if your application ships a file of URLs, you could have a second file that tracks URLs added by the user or reference URLs that were deleted by the user.

XML Resources

If, however, your file is in an XML format, you are better served not putting it in `res/raw/`, but rather in `res/xml/`. This is a directory for XML resources – resources known to be in XML format, but without any assumptions about what that XML represents.

To access that XML, you once again get a `Resources` object by calling `getResources()` on your `Activity` or other `Context`. Then, call `getXml()` on the `Resources` object, supplying the ID value of your XML resource (e.g., `R.xml.words`). This will return an `XmlResourceParser`, which implements the `XmlPullParser` interface. We will discuss how to use this parser, and the performance advantage of using XML resources, later in this chapter.

As with raw resources, XML resources are read-only at runtime.

Assets

Your third option is to package the data in the form of an asset. You can create an `assets/` directory at the root of your project directory, then place whatever files you want in there. Those are accessible at runtime by calling `getAssets()` on your

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`Activity` or other `Context`, then calling `open()` with the path to the file (e.g., `assets/foo/index.html` would be retrieved via `open("foo/index.html")`). As with raw resources, this returns an `InputStream` on the file's contents. And, as with all types of resources, assets are read-only at runtime.

One benefit of using assets over raw resources is the `file:///android_asset/ Uri` prefix. You can use this to load an asset into a `WebView`. For example, for an asset located in `assets/foo/index.html` within your project, calling `loadUrl("file:///android_asset/foo/index.html")` will load that HTML into the `WebView`.

Note that assets are compressed when the APK is packaged. Unfortunately, on Android 1.x/2.x, this compression mechanism has a 1MB file size limit. If you wish to package an asset that is bigger than 1MB, you either need to give it a file extension that will not be compressed (e.g., `.mp3`) or actually store a ZIP file of the asset (to avoid the automatic compression) and decompress it yourself at runtime, using the standard `java.util.zip` classes. This restriction was lifted with Android 3.0, and so if your `minSdkVersion` is 11 or higher, this will not be an issue for you.

Files and Android

On the whole, Android just uses normal Java file I/O for local files. You will use the same `File` and `InputStream` and `OutputWriter` and other classes that you have used time and again in your prior Java development work.

What is distinctive in Android is *where* you read and write. Akin to writing a Java Web app, you do not have read and write access to arbitrary locations. Instead, there are only a handful of directories to which you have any access, particularly when running on production hardware.

Internal vs. External

Internal storage refers to your application's portion of the on-board, always-available flash storage. External storage refers to storage space that can be mounted by the user as a drive in Windows (or, possibly with some difficulty, as a volume in OS X or Linux).

Historically (i.e., Android 1.x/2.x), internal storage was very limited in space. That is far less of a problem on 3.0 and higher.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Similarly, external storage is not always available on Android 1.x and 2.x – if it is mounted as a drive or volume on a host desktop or notebook, your app will not have access to external storage. We will examine this limitation in a bit more detail later in this chapter. This is not usually a problem on Android 3.0+.

Standard vs. Cache

On both internal and external storage, you have the option of saving files as a cache, or on a more permanent basis. Files located in a cache directory may be deleted by the OS or third-party apps to free up storage space for the user. Files located outside of cache will remain unless manually deleted.

Yours vs. Somebody Else's

Internal storage is on a per-application basis. Files you write to in your own internal storage cannot be read or written to by other applications... normally. Users who "root" their phones can run apps with superuser privileges and be able to access your internal storage. Most users do not root their phones, and so only your app will be able to access your internal storage files.

Files on external storage, though, are visible to all applications and the user. Anyone can read anything stored there, and any application that requests to can write or delete anything it wants.

Working with Internal Storage

You have a few options for manipulating the contents of your app's portion of internal storage.

One possibility is to use `openFileInput()` and `openFileOutput()` on your `Activity` or other `Context` to get an `InputStream` and `OutputStream`, respectively. However, these methods do not accept file paths (e.g., `path/to/file.txt`), just simple filenames.

If you want to have a bit more flexibility, `getFilesDir()` and `getCacheDir()` return a `File` object pointing to the roots of your files and cache locations on internal storage, respectively. Given the `File`, you can create files and subdirectories as you see fit.

To see how this works, take a peek at the `Files/ReadWrite` sample project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This application implements an `EditorFragment`, containing a full-screen `EditText`, hosted by a `FilesDemoActivity` as a static fragment. There is a `CheckBox` in the action bar to toggle between using internal and external storage:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/location"
    android:actionLayout="@layout/action_location"
    android:showAsAction="always">
  </item>
  <item
    android:id="@+id/save"
    android:icon="@android:drawable/ic_menu_save"
    android:showAsAction="always|withText"
    android:title="@string/save">
  </item>
  <item
    android:id="@+id/saveBackground"
    android:icon="@android:drawable/ic_menu_save"
    android:showAsAction="never"
    android:title="@string/saveBackground">
  </item>

</menu>
```

We get at that `CheckBox` in `onCreateOptionsMenu()` of `EditorFragment`, storing it in a data member of the fragment:

```
  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.actions, menu);
    external=(CheckBox)menu.findItem(R.id.location).getActionView();
  }
```

When they go to work with the file (e.g., press a Save toolbar button), we use a `getTarget()` method to return a `File` object pointing at the file to be manipulated. In the case where the `CheckBox` is unchecked — meaning we are to use internal storage — `getTarget()` uses `getFilesDir()`:

```
  private File getTarget() {
    File root=null;

    if (external != null && external.isChecked()) {
      root=getActivity().getExternalFilesDir(null);
    }
    else {
      root=getActivity().getFilesDir();
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(new File(root, FILENAME));
  }
```

Note that the `CheckBox` may not yet exist, depending on the timing of `onCreateOptionsMenu()` and `onResume()`. That timing varies a bit by Android version.

Methods like `load()` then load that `File` by using standard Java file I/O:

```
  private String load(File target) throws IOException {
    String result="";

    try {
      InputStream in=new FileInputStream(target);

      if (in != null) {
        try {
          InputStreamReader tmp=new InputStreamReader(in);
          BufferedReader reader=new BufferedReader(tmp);
          String str;
          StringBuilder buf=new StringBuilder();

          while ((str=reader.readLine()) != null) {
            buf.append(str);
            buf.append("\n");
          }

          result=buf.toString();
        }
        finally {
          in.close();
        }
      }
    }
    catch (java.io.FileNotFoundException e) {
      // that's OK, we probably haven't created it yet
    }

    return(result);
  }
```

The files stored in internal storage are accessible only to your application, by default. Other applications on the device have no rights to read, let alone write, to this space. However, bear in mind that some users "root" their Android phones, gaining superuser access. These users will be able to read and write whatever files they wish. As a result, please consider application-local files to be secure against malware but not necessarily secure against interested users.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Working with External Storage

On most Android 1.x devices and some early Android 2.x devices, external storage came in the form of a micro SD card or the equivalent. On the remaining Android 2.x devices, external storage was part of the on-board flash, but housed in a separate partition from the internal storage. On most Android 3.0+ devices, external storage is now simply a special directory in the partition that holds internal storage.

Devices will have at least 1GB of external storage free when they ship to the user. That being said, many devices have much more than that, but the available size at any point could be smaller than 1GB, depending on how much data the user has stored.

Where to Write

If you have files that are tied to your application that are simply too big to risk putting in internal storage, or if the user should be able to download the files off their device at will, you can use `getExternalFilesDir()`, available on any activity or other `Context`. This will give you a `File` object pointing to an automatically-created directory on external storage, unique for your application. While not secure against other applications, it does have one big advantage: when your application is uninstalled, these files are automatically deleted, just like the ones in the application-local file area. This method was added in API Level 8. This method takes one parameter — typically `null` — that indicates a particular type of file you are trying to save (or, later, load).

For example, the aforementioned `getTarget()` method of `EditorFragment` uses `getExternalFilesDir()` if the user has checked the `CheckBox` in the action bar:

```
private File getTarget() {
  File root=null;

  if (external != null && external.isChecked()) {
    root=getActivity().getExternalFilesDir(null);
  }
  else {
    root=getActivity().getFilesDir();
  }

  return(new File(root, FILENAME));
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

There is also `getExternalCacheDir()`, which returns a `File` pointing at a directory that contains files that you would like to have, but if Android or a third-party app clears the cache, your app will continue to function normally.

Android 4.4 (API Level 19) added two new methods, `getExternalCacheDirs()` and `getExternalFileDirs()`, the plural versions of the classic methods. These return an array of `File` objects, representing one *or more* places where your app can work with external storage. The first element in the array will be the same `File` object returned by the singular versions of the methods (e.g., `getExternalFilesDir()`). The other elements in the array, if any, will represent app-specific directories on alternative external storage locations, like removable cards. The Android Support package has a `ContextCompat` class containing static versions of `getExternalCacheDirs()` and `getExternalFileDirs()`, so you can use the same code on API Level 4 and above, though the backport will only ever return one directory in the array.

If you have files that belong more to the user than to your app — pictures taken by the camera, downloaded MP3 files, etc. — a better solution is to use `getExternalStoragePublicDirectory()`, available on the `Environment` class. This will give you a `File` object pointing to a directory set aside for a certain type of file, based on the type you pass into `getExternalStoragePublicDirectory()`. For example, you can ask for `DIRECTORY_MOVIES`, `DIRECTORY_MUSIC`, or `DIRECTORY_PICTURES` for storing MP4, MP3, or JPEG files, respectively. These files will be left behind when your application is uninstalled. This method was also added in API Level 8.

You will also find a `getExternalStorageDirectory()` method on `Environment`, pointing to the root of the external storage. This is no longer the preferred approach — the methods described above help keep the user's files better organized. However, if you are supporting older Android devices, you may need to use `getExternalStorageDirectory()`, simply because the newer options may not be available to you.

Relevant Permissions

On all relevant Android versions prior to Android 4.4 (API Level 19), if you want to write to external storage, you need to hold the `WRITE_EXTERNAL_STORAGE` permission. And, on those versions, you do not need a permission to read from external storage.

On Android 4.4 and up, the rules are a bit different:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- To read or write in the directory trees rooted at `getExternalFilesDir()` and `getExternalCacheDir()`, you do not need a permission
- To write to anywhere else on external storage, you need `WRITE_EXTERNAL_STORAGE`
- To read from anywhere else on external storage, you need either `WRITE_EXTERNAL_STORAGE` (if you already have that) or `READ_EXTERNAL_STORAGE` (if not)

Hence, so long as your `android:minSdkVersion` is less than 19, you need to take the most conservative approach:

- If you are writing *anywhere* on external storage, request the `WRITE_EXTERNAL_STORAGE` permission
- If you are only reading, but from *anywhere* on external storage, request the `READ_EXTERNAL_STORAGE` permission

Note that you might get paths to external storage locations from third-party apps, typically in the form of a `Uri`. If you are handling `Uri` values from third-party apps, you should request `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE`, in case the third-party app hands you a `Uri` pointing to external storage.

For example, here is the sample app's manifest, complete with the `<uses-permission>` element for `WRITE_EXTERNAL_STORAGE`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.frw"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="14"/>

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"
    android:xlargeScreens="true"/>

  <uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE"
    android:maxSdkVersion="18"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:theme="@android:style/Theme.Holo.Light.DarkActionBar"
    android:uiOptions="splitActionBarWhenNarrow">
    <activity
      android:name=".FilesDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Note the use of `android:maxSdkVersion`. This tells Android to only request this permission up to the stated API level. After that, ignore the request. This sample uses `getExternalFilesDir()` and therefore does not need `WRITE_EXTERNAL_STORAGE` on API Level 19+ devices. Users who install this app on such devices will not be prompted for this permission, while users who install this app on older devices will.

When to Write

Also, external storage may be tied up by the user having mounted it as a USB storage device. You can use `getExternalStorageState()` (a static method on `Environment`) to determine if external storage is presently available or not. On Android 3.0 and higher, this should be much less of an issue, as they changed how the external storage is used by the host PC — originally, this used USB Mass Storage Mode (think thumb drives) and now uses the USB Media Transfer Protocol (think MP3 players). With MTP, both the Android device and the PC it is connected to can have access to the files simultaneously; Mass Storage Mode would only allow the host PC to have access to the files if external storage is mounted.

Nowadays, you can use `getStorageState()` on the `Environment` class (or `getStorageState()` on the `EnvironmentCompat` class from the Android Support package) to find out the state of external storage, for the particular `File` passed as a parameter.

Letting the User See Your Files

The switch to MTP has one side-effect for Android developers: files you write to external storage may not be automatically visible to the user. At the time of this writing, the only files that will show up on the user's PC will be ones that have been indexed by the `MediaStore`. While the `MediaStore` is typically thought of as only

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

indexing "media" (images, audio files, video files, etc.), it was given the added role in Android 3.0 of maintaining an index of *all* files for the purposes of MTP.

Your file that you place on external storage will not be indexed automatically simply by creating it and writing to it. Eventually, it will be indexed, though it may be quite some time for an automatic indexing pass to take place.

To force Android to index your file, you can use `scanFile()` on `MediaScannerConnection`:

```
String[] paths={pathToYourNewFileOnExternalStorage};
MediaScannerConnection.scanFile(this, paths, null, null);
```

The third parameter to `scanFile()` is an array of MIME types, to line up with the array of paths in the second parameter. If your file is some form of media, and you know the MIME type, supplying that will ensure that your media will be visible as appropriate to the right apps (e.g., images in the Gallery app). Otherwise, Android will try to infer a MIME type from the file extension.

Limits on External Storage Open Files

Many Android devices will have a per-process limit of 1024 open files, on any sort of storage. This is usually not a problem for developers.

On some devices — including probably all that are running Android 4.2 and higher — there is a *global* limit of 1024 open files on external storage. In other words, all running apps combined can only open 1024 files simultaneously on external storage.

This means that it is important for you to minimize how many open files on external storage you have at a time. Having a few open files is perfectly reasonable; having a few hundred open files is not.

Multiple User Accounts

On Android 4.1 and earlier, each Android device was assumed to be used by just one person.

On Android 4.2 and higher, though, it is possible for a tablet owner to set up multiple user accounts. Each user gets their own section of internal and external storage for files, databases, `SharedPreferences`, and so forth. From your standpoint,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

it is as if the users are really on different devices, even though in reality it is all the same hardware.

However, this means that paths to internal and external storage now may vary by user. Hence, it is *very important* for you to use the appropriate methods, outlined in this chapter, for finding locations on internal storage (e.g., `getFilesDir()`) and external storage (e.g., `getExternalFilesDir()`).

Some blog posts, Stack Overflow answers, and the like will show the use of hard-coded paths for these locations (e.g., `/sdcard` or `/mnt/sdcard` for the root of external storage). Hard-coding such paths was never a good idea. And, as of Android 4.2, those paths are simply wrong and will not work.

On Android 4.2+, for the original user of the device, internal storage will wind up in the same location as before, but external storage will use a different path. For the second and subsequent users defined on the device, both internal and external storage will reside in different paths. The various methods, like `getFilesDir()`, will handle this transparently for you.

Note that, at the time of this writing, multiple accounts are not available on the emulators, only on actual tablets. Phones usually will not have multiple-account support, under the premise that tablets are more likely to be shared than are phones.

Linux Filesystems: You Sync, You Win

Android is built atop a Linux kernel and uses Linux filesystems for holding its files. Classically, Android used YAFFS (Yet Another Flash File System), optimized for use on low-power devices for storing data to flash memory.

YAFFS has one big problem: only one process can write to the filesystem at a time. For those of you into filesystems, rather than offering file-level locking, YAFFS has partition-level locking. This can become a bit of a bottleneck, particularly as Android devices grow in power and start wanting to do more things at the same time like their desktop and notebook brethren.

Android 3.0 switched to ext4, another Linux filesystem aimed more at desktops/notebooks. Your applications will not directly perceive the difference. However, ext4 does a fair bit of buffering, and it can cause problems for applications that do not take this buffering into account. Linux application developers ran headlong into this

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

in 2008–2009, when ext4 started to become popular. Android developers will need to think about it now... for your own file storage.

If you are using SQLite or `SharedPreferences`, you do not need to worry about this problem. Android (and SQLite, in the case of SQLite) handle all the buffering issues for you. If, however, you write your own files, you may wish to contemplate an extra step as you flush your data to disk. Specifically, you need to trigger a Linux system call known as `fsync()`, which tells the filesystem to ensure all buffers are written to disk.

If you are using `java.io.RandomAccessFile` in a synchronous mode, this step is handled for you as well, so you will not need to worry about it. However, Java developers tend to use `FileOutputStream`, which does not trigger an `fsync()`, even when you call `close()` on the stream. Instead, you call `getFD().sync()` on the `FileOutputStream` to trigger the `fsync()`. Note that this may be time-consuming, and so disk writes should be done off the main application thread wherever practical, such as via an `AsyncTask`.

This is why, in `EditorFragment`, our `save()` implementation looks like this:

```
  private void save(String text, File target) throws IOException {
    FileOutputStream fos=new FileOutputStream(target);
    OutputStreamWriter out=new OutputStreamWriter(fos);

    out.write(text);
    out.flush();
    fos.getFD().sync();
    out.close();
  }
```

StrictMode: Avoiding Janky Code

Users are more likely to like your application if, to them, it feels responsive. Here, by "responsive", we mean that it reacts swiftly and accurately to user operations, like taps and swipes.

Conversely, users are less likely to be happy with you if they perceive that your UI is "janky" — sluggish to respond to their requests. For example, maybe your lists do not scroll as smoothly as they would like, or tapping a button does not yield the immediate results they seek.

While threads and `AsyncTask` and the like can help, it may not always be obvious where you should be applying them. A full-scale performance analysis, using

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Traceview or similar Android tools, is certainly possible. However, there are a few standard sorts of things that developers do, sometimes quite by accident, on the main application thread that will tend to cause sluggishness:

1. Flash I/O, both for internal and external storage
2. Network I/O

However, even here, it may not be obvious that you are performing these operations on the main application thread. This is particularly true when the operations are really being done by Android's code that you are simply calling.

That is where `StrictMode` comes in. Its mission is to help you determine when you are doing things on the main application thread that might cause a janky user experience.

`StrictMode` works on a set of policies. There are presently two categories of policies: VM policies and thread policies. The former represent bad coding practices that pertain to your entire application, notably leaking SQLite `Cursor` objects and kin. The latter represent things that are bad when performed on the main application thread, notably flash I/O and network I/O.

Each policy dictates what `StrictMode` should watch for (e.g., flash reads are OK but flash writes are not) and how `StrictMode` should react when you violate the rules, such as:

1. Log a message to LogCat
2. Display a dialog
3. Crash your application (seriously!)

The simplest thing to do is call the static `enableDefaults()` method on `StrictMode` from `onCreate()` of your first activity. This will set up normal operation, reporting all violations by simply logging to LogCat. However, you can set your own custom policies via `Builder` objects if you so choose.

However, do not use `StrictMode` in production code. It is designed for use when you are building, testing, and debugging your application. It is not designed to be used in the field.

In `FilesDemoActivity`, in addition to loading `R.layout.main` with our `EditorFragment` statically defined, we configure `StrictMode`, if and only if we are

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

building a debug version of the app and are on a version of Android that supports `StrictMode`:

```
package com.commonsware.android.frw;

import android.app.Activity;
import android.os.Build;
import android.os.Bundle;
import android.os.StrictMode;

public class FilesDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    if (BuildConfig.DEBUG
        && Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD) {
      StrictMode.setThreadPolicy(buildPolicy());
    }
  }

  private StrictMode.ThreadPolicy buildPolicy() {
    return(new StrictMode.ThreadPolicy.Builder().detectAll()
                                                .penaltyLog().build());
  }
}
```

Here, we are asking to flag all faults (`detectAll()`), logging any violations to LogCat (`penaltyLog()`).

`BuildConfig.DEBUG` is a flag that indicates if we are on a debug build or not. The `BuildConfig` class is code-generated alongside the `R` class, in whatever Java package we declared in the `<manifest>` element of our manifest.

If we press the "Save" action bar item, instead of going to the menu and using "Save in Background", we will do disk I/O on the main application thread and generate `StrictMode` violations as a result:

```
D/StrictMode(1902): StrictMode policy violation; ~duration=1 ms:
android.os.StrictMode$StrictModeDiskWriteViolation: policy=31 violation=1
D/StrictMode(1902):   at
android.os.StrictMode$AndroidBlockGuardPolicy.onWriteToDisk(StrictMode.java:1097)
D/StrictMode(1902):   at libcore.io.BlockGuardOs.fsync(BlockGuardOs.java:96)
D/StrictMode(1902):   at java.io.FileDescriptor.sync(FileDescriptor.java:74)
D/StrictMode(1902):   at
com.commonsware.android.frw.EditorFragment.save(EditorFragment.java:111)
...
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

While wordy, and logged only at debug severity, this is enough to point out where in your code the violation occurred — in our case, in `onOptionsItemSelected()` of `EditorFragment`.

Note that `StrictMode` will also report leaked open files. For example, if you create a `FileOutputStream` on a `File` and fail to `close()` it later, when the `FileOutputStream` (and related objects) are garbage-collected, `StrictMode` will report to you the fact that you failed to close the stream. This is very useful to help you make sure that you are not leaking open files that may contribute to exhausting the 1,024 open file limit on external storage.

XML Parsing Options

Android supports a fairly standard implementation of the Java DOM and SAX APIs. If you have existing experience with these, or if you have code that already leverages them, feel free to use them.

Android also bakes in the `XmlPullParser` from the xmlpull.org site. Like SAX, the `XmlPullParser` is an event-driven interface, compared to the DOM that builds up a complete data structure and hands you that result. Unlike SAX, which relies on a listener and callback methods, the `XmlPullParser` has you pull events off a queue, ignoring those you do not need and dispatching the rest as you see fit to the rest of your code.

The primary reason the `XmlPullParser` was put into Android was for XML-encoded resources. While you write plain-text XML during development, what is packaged in your APK file is a so-called "binary XML" format, where angle brackets and quotation marks and such are replaced by bitfields. This helps compression a bit, but mostly this conversion is done to speed up parsing. Android's XML resource parser can parse this "binary XML" approximately ten times faster than it can parse the equivalent plain-text XML. Hence, anything you put in an XML resource (`res/xml/`) will be parsed similarly quickly.

For plain-text XML content, the `XmlPullParser` is roughly equivalent, speed-wise, to SAX. All else being equal, lean towards SAX, simply because more developers will be familiar with it from classic Java development. However, if you really like the `XmlPullParser` interface, feel free to use it.

You are welcome to try a third-party XML parser JAR, but bear in mind that there may be issues when trying to get it working in Android.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

JSON Parsing Options

Android has bundled the `org.json` classes into the SDK since the beginning, for use in parsing JSON. These classes have a DOM-style interface: you hand `JSONObject` a hunk of JSON, and it gives you an in-memory representation of the completely parsed result. This is handy but, like the DOM, a bit of a performance hog.

API Level 11 added `JSONReader`, based on Google's GSON parser, as a "streaming" parser alternative. `JSONReader` is much more reminiscent of the `XmlPullParser`, in that you pull events out of the "reader" and process them. This can have significant performance advantages, particularly in terms of memory consumption, if you do not need the entire JSON data structure. However, this is only available on API Level 11 and higher.

Because `JSONReader` is a bit "late to the party", there has been extensive work on getting other JSON parsers working on Android. Google's GSON is popular, as is Jackson. Jackson offers a few APIs, and the streaming API reportedly works very nicely on Android with top-notch performance.

Visit the Trails!

In addition to this chapter, you can learn more about accessing multimedia files via the `MediaStore` and learn more about the impacts of multiple user accounts on tablets.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #11 - Adding Simple Content

Now that we have seen how to work with assets, we can start putting them to use, by defining some "help" and "about" HTML files and displaying them in their respective activities.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Adding Some Content

Your project may already have an `assets/` folder. If not, create one:

- In Android Studio, right-click over the `main` sourceset directory, choose New > Directory from the context menu, fill in the name `assets` in the dialog, and click OK
- In Eclipse, right-click over the project in the Package Explorer, choose New > Folder from the context menu, fill in the name `assets` in the dialog, and click Finish

In `assets/`, create a `misc/` sub-folder, by right-clicking over the `assets/` folder and choosing to add a new directory named `misc` (e.g., New > Directory from the Android Studio context menu).

In `assets/misc/`, create two files, `about.html` and `help.html`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- In Android Studio, right-click over the `assets/misc/` folder in the project explorer, choose New > File from the context menu, fill in the desired filename in the dialog, and click OK
- Eclipse users can create files by right-clicking over the folder, choosing New > File from the context menu, supplying the name of the file, and clicking "Finish".

The actual HTML content of these two files does not matter, so long as you can tell them apart when looking at them. If you prefer, you can download sample `about.html` and `help.html` files from the application's GitHub repository, via the links.

Eclipse users should note that the default behavior of double-clicking on an HTML file in the IDE is to open it in some Eclipse-supplied internal browser. This is not especially useful. If you right-click over the file and choose Open With > Text Editor, from that point forward that specific file will be opened in an editor pane you can use to add or edit the HTML you want to have.

Step #2: Using SimpleContentFragment

Now, open up `SimpleContentActivity` and replace the stub implementation that we have now with the following Java:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.Fragment;
import android.os.Bundle;

public class SimpleContentActivity extends Activity {
  public static final String EXTRA_FILE="file";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      String file=getIntent().getStringExtra(EXTRA_FILE);
      Fragment f=SimpleContentFragment.newInstance(file);

      getFragmentManager().beginTransaction()
                          .add(android.R.id.content, f).commit();
    }
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

In `onCreate()`, we follow the standard recipe for defining our fragment if (and only if) we were started new, rather than restarted after a configuration change, by seeing if the fragment already exists. If we do need to add the fragment, we retrieve a string extra from the `Intent` used to launch us (identified as `EXTRA_FILE`), create an instance of `SimpleContentFragment` using that value from the extra, and execute a `FragmentTransaction` to add the `SimpleContentFragment` to our UI.

Step #3: Launching Our Activities, For Real This Time

Now, what remains is to actually supply that `EXTRA_FILE` value, which we are not doing presently when we start up `SimpleContentActivity` from `EmPubLiteActivity`.

Modify `onOptionsItemSelected()` of `EmPubLiteActivity` to look like this:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case android.R.id.home:
      return(true);

    case R.id.about:
      Intent i=new Intent(this, SimpleContentActivity.class);

      i.putExtra(SimpleContentActivity.EXTRA_FILE,
                 "file:///android_asset/misc/about.html");
      startActivity(i);

      return(true);

    case R.id.help:
      i=new Intent(this, SimpleContentActivity.class);

      i.putExtra(SimpleContentActivity.EXTRA_FILE,
                 "file:///android_asset/misc/help.html");
      startActivity(i);

      return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You are adding the two `putExtra()` calls in the `R.id.about` and `R.id.help` branches of the `switch` statement. In both cases, we are using a quasi-URL with the prefix `file:///android_asset/`. This points to the root of our project's `assets/` folder. `WebView` knows how to interpret these URLs, to load files out of our assets directly.

Now, if you run the application and choose "Help" from the action bar overflow, you will see your help content on-screen:

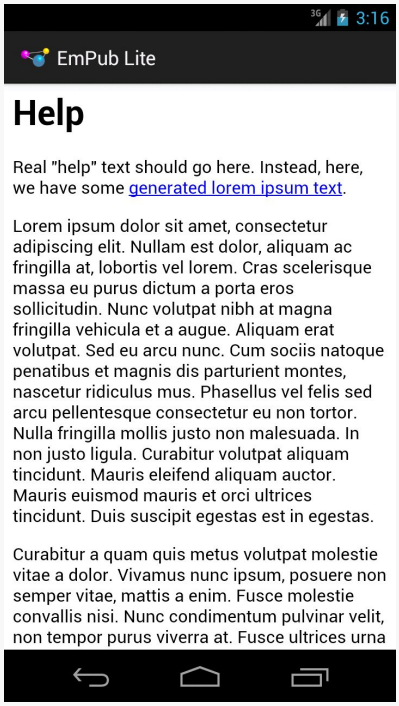


Figure 259: EmPubLite Help Screen

Pressing BACK and choosing "About" from the action bar overflow will bring up your about content:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

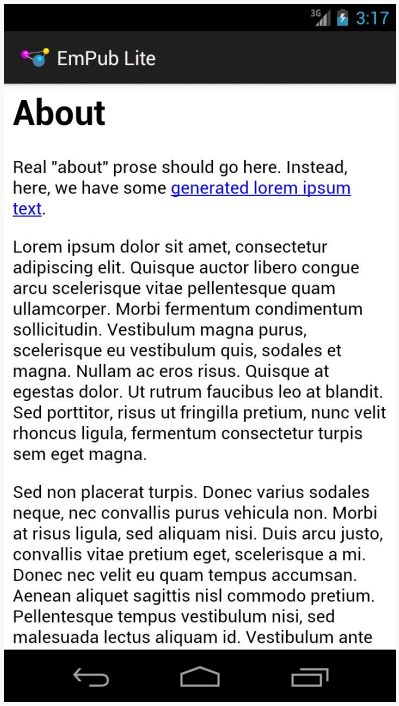


Figure 260: EmPubLite About Screen

In Our Next Episode…

… we will display the actual content of our book in our tutorial project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #12 - Displaying the Book

At this point, you are probably wondering when we are *ever* going to have our digital book reader let us read a digital book.

Now, in this tutorial, your patience will be rewarded.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Note that starting in this tutorial, it is assumed that you know how to add `import` statements as needed as we refer to new classes in existing code, and so the required imports are not always going to be specified.

Step #1: Adding a Book

First, we need a book. Expecting you to write a book as part of this tutorial would seem to be a bit excessive. So, instead, we will use an already-written book: The War of the Worlds, by H. G. Wells, as distributed by Project Gutenberg.

EDITOR'S NOTE: We realize that this choice of book may be seen as offensive by Martians, as it depicts them as warlike invaders with limited immune systems. Please understand that this book is a classic of Western literature and reflects the attitude of the times. If you have any concerns about this material, please contact us at *martians-so-do-not-exist@commonsware.com*.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Download http://misc.commonsware.com/WarOfTheWorlds.zip and unpack its contents (a `book/` directory of files) into your `assets/` folder of your project. IDE users can drag this `book/` directory into the project and drop it in `assets/` to copy the files to the proper location. You should wind up with `assets/book/` and files inside of there.

In that directory, you will find some HTML and CSS files with the prose of the book, plus a `contents.json` file with metadata. We will examine this metadata in greater detail in the next section.

Step #2: Creating a ModelFragment

This sample project will use the "model fragment" pattern to hold onto the data about the book to be viewed. The "model fragment" pattern works well for cases where:

- the data is only needed by one activity, not several components, and
- we want to hold onto the data during a configuration change (e.g., screen rotation), so that we do not have to perform some work again to obtain the data

Something has to load that `BookContents`, ideally in the background, since reading an asset and parsing the JSON will take time. Also, something has to hold onto that `BookContents`, so it can be used from `EmPubLiteActivity` and the various chapter fragments in the `ViewPager`.

To that end, we will create a new class, cunningly named `ModelFragment`.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `ModelFragment` as the name and click OK to create the empty class.

Then, replace the contents of that class with the following:

```
package com.commonsware.empublite;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.app.Fragment;

public class ModelFragment extends Fragment {

}
```

Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. Fill in `ModelFragment` in the "Name" field. Then, click the "Browse..." button next to the "Superclass" field and find `android.app.Fragment` to set as the superclass. Then, click "Finish" on the new-class dialog to create the `ModelFragment` class.

Step #3: Defining Our Model

That `contents.json` file contains a bit of metadata about the contents of the book: the book's title and a roster of its "chapters":

```
{
  "title": "The War of the Worlds",
  "chapters": [
    {
      "file": "0.htm",
      "title": "Book One: Chapters 1-9"
    },
    {
      "file": "1.htm",
      "title": "Book One: Chapters 10-14"
    },
    {
      "file": "2.htm",
      "title": "Book One: Chapters 14-17"
    },
    {
      "file": "3.htm",
      "title": "Book Two: Chapters 1-7"
    },
    {
      "file": "4.htm",
      "title": "Book Two: Chapters 7-10"
    },
    {
      "file": "5.htm",
      "title": "Project Gutenberg"
    }
  ]
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In the case of this book from Project Gutenberg, the `assets/book/` directory contains six HTML files which `EmPubLite` will consider as "chapters", even though each of those HTML files contains multiple chapters from the source material. You are welcome to reorganize that HTML if you wish, updating `contents.json` to match.

We need to load `contents.json` into memory, so `EmPubLite` knows how many chapters to display and where those chapters can be found. We will pour `contents.json` into a `BookContents` model object, leveraging the GSON library that we added to our project in an earlier tutorial.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `BookContents` as the name and click OK to create the empty class.

Then, replace the contents of that class with the following:

```
package com.commonsware.empublite;

import java.util.List;

public class BookContents {
  String title;
  List<BookContents.Chapter> chapters;

  int getChapterCount() {
    return(chapters.size());
  }

  String getChapterFile(int position) {
    return(chapters.get(position).file);
  }

  String getTitle() {
    return(title);
  }

  static class Chapter {
    String file;
    String title;
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. Fill in `BookContents` in the "Name" field. Leave the "Superclass" field alone, as `BookContents` has no explicit superclass. Then, click "Finish" on the new-class dialog to create the `BookContents` class.

Then, with `BookContents` open in the editor, paste in the class definition shown in the Android Studio instructions above.

Step #4: Examining Our Model

`BookContents` is a GSON interpretation of the JSON structure of `contents.json`. `BookContents` holds onto the `title` and `chapters`, with the latter being a `List` of `BookContents.Chapter` objects, each of which holds onto its file and title.

`BookContents` also supplies three accessor methods:

- `getChapterCount()` to identify the number of chapters (i.e., the size of the `chapters` array in the JSON)
- `getChapterFile()`, to return the relative path within `assets/book/` that represents our "chapter" of HTML
- `getTitle()` to retrieve the book title out of the object

Step #5: Defining Our Event

We will want to load the JSON and create the `BookContents` on a background thread, as we will be performing enough I/O and parsing that we might make our UI a bit sluggish if we do the work on the main application thread. However, we need to let the UI layer (`EmPubLiteActivity` and its `ViewPager`) know when the book is loaded, so it can be poured into the user interface.

We could use an `AsyncTask` for that, notifying the activity in `onPostExecute()`. However, we will need more flexible inter-component communication over time, things that cannot be handled by a simple `AsyncTask`. Hence, we will start using the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

event bus pattern here, employing greenrobot's EventBus library that we added to our project in a previous tutorial.

With EventBus, we create our own event classes. The one event that we have up front is one to indicate that our book metadata has been loaded and is ready for use, in the form of a `BookContents` object. Hence, in this step of the tutorial, we will define a `BookLoadedEvent` that will be posted when the book is loaded. And, we will have the event hold onto the `BookContents`, to lightly simplify populating the UI later on.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `BookLoadedEvent` as the name and click OK to create the empty class.

Then, replace the contents of that class with the following:

```
package com.commonsware.empublite;

public class BookLoadedEvent {
  private BookContents contents=null;

  public BookLoadedEvent(BookContents contents) {
    this.contents=contents;
  }

  public BookContents getBook() {
    return(contents);
  }
}
```

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. Fill in `BookLoadedEvent` in the "Name" field. Leave the "Superclass" field alone, as `BookLoadedEvent` has no

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

explicit superclass. Then, click "Finish" on the new-class dialog to create the `BookLoadedEvent` class.

Then, with `BookLoadedEvent` open in the editor, paste in the class definition shown in the Android Studio instructions above.

Step #6: Loading Our Model

Now, we need to actually arrange to load the book on a background thread and post our newly-created `BookLoadedEvent`. This is one of the key jobs of our `ModelFragment`: to manage the loading of our activity's model, using background threads.

With that in mind, replace our stub `ModelFragment` implementation with the following:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.Fragment;
import android.content.res.AssetManager;
import android.os.Bundle;
import android.os.Process;
import android.util.Log;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import com.google.gson.Gson;
import de.greenrobot.event.EventBus;

public class ModelFragment extends Fragment {
  private BookContents contents=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setRetainInstance(true);
  }

  @Override
  public void onAttach(Activity host) {
    super.onAttach(host);

    if (contents == null) {
      new LoadThread(host.getAssets()).start();
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }

  public BookContents getBook() {
    return(contents);
  }

  private class LoadThread extends Thread {
    private AssetManager assets=null;

    LoadThread(AssetManager assets) {
      super();

      this.assets=assets;
      Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
    }

    @Override
    public void run() {
      Gson gson=new Gson();

      try {
        InputStream is=assets.open("book/contents.json");
        BufferedReader reader=
            new BufferedReader(new InputStreamReader(is));

        contents=gson.fromJson(reader, BookContents.class);
        EventBus.getDefault().post(new BookLoadedEvent(contents));
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
      }
    }
  }
}
```

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

In `onCreate()`, we call `setRetainInstance(true)`, to tell the framework to keep this fragment despite a configuration change, just passing it to the new activity created as a result of that configuration change.

In `onAttach()`, if we do not already have our `BookContents` object, we fork a `LoadThread` to populate it, and we cannot readily get at an `AssetManager` until we are attached to the hosting activity. This is why we are not forking `LoadThread` in `onCreate()`.

`LoadThread` takes the `AssetManager` as a parameter, stashing it in a data member in the `LoadThread` constructor. The constructor also calls `setThreadPriority()` to drop

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the thread's priority to that of a background thread. This reduces how much we compete with the main application thread for CPU time.

Then, in the `run()` method that is called on the background thread, we read in the JSON using GSON to create the `BookContents` instance. GSON automatically de-serializes our JSON into the `BookContents` and `BookContents.Chapter` instances, given that we are telling the `fromJson()` method that it is to be loading an instance of a `BookContents` object. Finally, we `post()` a `BookLoadedEvent` to the default `EventBus`.

The `open()` method on `AssetManager` could throw an `IOException`. Normally, this indicates a development-time bug (e.g., we failed to actually set up the `book/contents.json` file), which is why we log the message to LogCat. A production-grade book reader should also `post()` an `EventBus` event to allow the UI layer to let the user know that we could not load the book. As it stands, the book reader will remain stuck on the `ProgressBar` forever in case of this sort of problem. Augmenting the tutorial in this way is left as an exercise for the reader.

Note that the `LoadThread` implementation has a pair of references to `Process`. In this case, this is `android.os.Process`, not `java.lang.Process`. Since `java.lang.Process` is automatically imported, if you fail to import `android.os.Process`, you will see errors about how `THREAD_PRIORITY_BACKGROUND` and `setThreadPriority()` are not defined. Since we are not using `java.lang.Process` in this class, having the import to `android.os.Process` (as shown in the code listing above) resolves this conflict.

Step #7: Registering for Events

Right now, our `BookLoadedEvent` will be posted... and ignored, as nothing in the application is set up to watch for such events. Our `EmPubLiteActivity` needs to know about these events, and the first step to accomplishing that is to have it register for events in general with the `EventBus`.

Add the following two methods to `EmPubLiteActivity`:

```
@Override
  public void onResume() {
    super.onResume();
    EventBus.getDefault().register(this);
  }

  @Override
  public void onPause() {
    EventBus.getDefault().unregister(this);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onPause();
  }
```

These simply register the activity with the `EventBus` while it is in the foreground.

Step #8: Adapting the Content

Before we can use the `BookContents`, we need to update `ContentsAdapter` to display the prose on the screen.

First, add a `BookContents` data member to `ContentsAdapter`:

```
  private BookContents contents=null;
```

Then, add the `BookContents` parameter to the constructor, assigning it to the new data member:

```
  public ContentsAdapter(Activity ctxt, BookContents contents) {
    super(ctxt.getFragmentManager());

    this.contents=contents;
  }
```

Next, update `getCount()` to use the `getChapterCount()` of our `BookContents`:

```
  @Override
  public int getCount() {
    return(contents.getChapterCount());
  }
```

Finally, modify `getItem()` to retrieve the relative path for a given chapter from the `BookContents` and create a `SimpleContentFragment` on the complete `file:///android_asset` path to the file in question:

```
  @Override
  public Fragment getItem(int position) {
    String path=contents.getChapterFile(position);

    return(SimpleContentFragment.newInstance("file:///android_asset/book/"
        + path));
  }
```

Note that you may need to change the parameter name in the `getItem()` declaration to be `position`, as it may be another value (e.g., `arg0`).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Step #9: Showing the Content When Loaded

Now, we can actually add the logic to display the book once it is loaded.

Create a `setupPager()` method on `EmPubLiteActivity` as follows:

```
  private void setupPager(BookContents contents) {
    adapter=new ContentsAdapter(this, contents);
    pager.setAdapter(adapter);
    findViewById(R.id.progressBar1).setVisibility(View.GONE);
    pager.setVisibility(View.VISIBLE);
  }
```

The contents of this method are almost identical to four lines in `onCreate()` – we have just moved them to a separate method. Remove those duplicate lines from `onCreate()`, so you have:

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    pager=(ViewPager)findViewById(R.id.pager);
  }
```

Then, add the following `onEventMainThread()` method to `EmPubLiteActivity`:

```
  public void onEventMainThread(BookLoadedEvent event) {
    setupPager(event.getBook());
  }
```

This tells `EventBus` that if a `BookLoadedEvent` is posted, we are interested in it, and it should be delivered to our `onEventMainThread()` method on the main application thread.

Step #10: Attaching our ModelFragment

We also need to add some code to set up the `ModelFragment` — it will not magically appear on its own. So, the first time we create an `EmPubLiteActivity`, we want to create our `ModelFragment`. To do that, define a static data member named `MODEL` in `EmPubLiteActivity`:

```
  private static final String MODEL="model";
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Then, add an `onResume()` method in `EmPubLiteActivity` to see if we already have the fragment before creating one:

```
@Override
public void onResume() {
  super.onResume();
  EventBus.getDefault().register(this);

  if (adapter==null) {
    ModelFragment mfrag=
        (ModelFragment)getFragmentManager().findFragmentByTag(MODEL);

    if (mfrag == null) {
      getFragmentManager().beginTransaction()
          .add(new ModelFragment(), MODEL).commit();
    }
    else if (mfrag.getBook() != null) {
      setupPager(mfrag.getBook());
    }
  }
}
```

If you run the result in a device or emulator, you will see the book content appear:

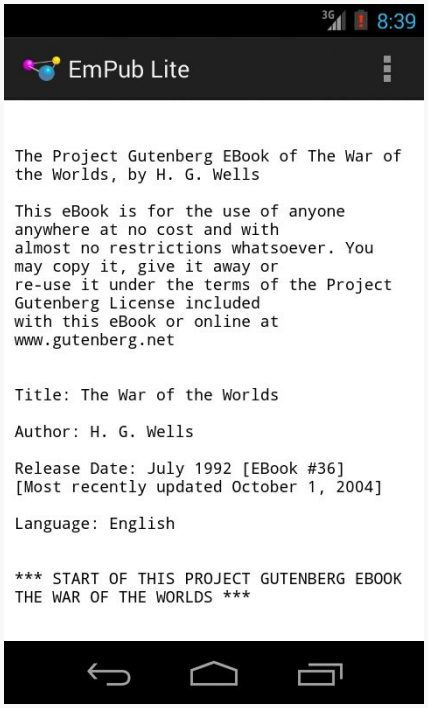


Figure 261: EmPubLite, With Content

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Swiping left and right will take you to the other portions of the book.

Step #11: Showing the Content After a Configuration Change

While you can see the book contents now, if you try rotating the screen, the book contents will not appear. That is because the `ModelFragment` has already loaded the contents (so the `BookLoadedEvent` has passed), but we have no logic in `EmPubLiteActivity` to populate the book by other means.

To do that, simply add an `else` clause to the `if` in `onResume()`, to get the book contents over to `setupPager()` if they are ready:

```
@Override
public void onResume() {
  super.onResume();
  EventBus.getDefault().register(this);

  if (adapter==null) {
    ModelFragment mfrag=
        (ModelFragment)getFragmentManager().findFragmentByTag(MODEL);

    if (mfrag == null) {
      getFragmentManager().beginTransaction()
          .add(new ModelFragment(), MODEL).commit();
    }
    else if (mfrag.getBook() != null) {
      setupPager(mfrag.getBook());
    }
  }
}
```

Now, if you run the sample and rotate the screen (e.g., `Ctrl-F11` on the Windows/Linux emulator), the book will appear in either case.

NOTE: Bugs in the Android 4.4 and Android 5.0 emulators prevent the app from finding out about the rotation event, and so nothing rotates. Please use an Android 4.3 or earlier emulator, or an actual Android device, for testing this.

Step #12: Going Home, Again

We can now take advantage of the icon in the action bar (a.k.a., the "app icon"), using it to bring us back to the first chapter. While the first chapter in our book is

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

not very distinctive compared to the other chapters, you might have a table of contents or a cover or something as the first "chapter".

The big thing that you need to do to support this is add one line to our `android.R.id.home` portion of the `switch()` in `onOptionsItemSelected()`, telling the `ViewPager` to go back to the first page:

```
      case android.R.id.home:
        pager.setCurrentItem(0, false);
        return(true);
```

You also need to teach the action bar to support taps on this "app icon", by adding this line to the bottom of your `onCreate()` of `EmPubLiteActivity`:

```
getActionBar().setHomeButtonEnabled(true);
```

If you run this on a device or emulator, swipe to some later chapter, then tap the icon in the upper-left corner, you will be returned to the first chapter.

Step #13: Setting Up StrictMode

Since we are now starting to do disk I/O, particularly aiming to have it done on background threads, it would be a good idea to configure `StrictMode`, so it will complain if we fail in our quest and accidentally do this I/O on the main application thread.

Add the following method to `EmPubLiteActivity`:

```
  private void setupStrictMode() {
    StrictMode.ThreadPolicy.Builder builder=
        new StrictMode.ThreadPolicy.Builder().detectNetwork();

    if (BuildConfig.DEBUG) {
      builder.penaltyDeath();
    }
    else {
      builder.penaltyLog();
    }

    StrictMode.setThreadPolicy(builder.build());
  }
```

Here, we create a `StrictMode.ThreadPolicy.Builder`, configured to detect network I/O on the main application thread. If we are in a `DEBUG` build, we will crash the app; otherwise, we will simply log the stack trace to LogCat.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You might wonder why we are not checking for disk I/O, since the opening paragraph of this section suggested that is what we would be doing. Ideally, we would have `detectDiskReads()` and `detectDiskWrites()` calls in here as well to enable those checks. Unfortunately, Google engineers do not check the framework code for these sorts of violations, leading to some bugs that we as app developers cannot resolve. As a result, for the purposes of this tutorial, we will not check for those problems.

Then, just after `super.onCreate()` in the `onCreate()` in `EmPubLiteActivity`, add in a call to the new `setupStrictMode()` method. This will give you an `onCreate()` method that looks like:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setupStrictMode();

  setContentView(R.layout.main);
  pager=(ViewPager)findViewById(R.id.pager);

  getActionBar().setHomeButtonEnabled(true);
}
```

If you set up an emulator running Android 5.0, please switch to an emulator running Android 4.3 or older. Android 5.0 has a bug which causes a `StrictMode` violation when we use our `WebViewFragment`.

Otherwise, if you run the app, nothing unusual should happen, which is good.

In Our Next Episode…

… we will allow the user to manipulate some preferences in our tutorial project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Using Preferences

Android has many different ways for you to store data for long-term use by your activity. The simplest ones to use are `SharedPreferences` and simple files.

Android allows activities and applications to keep preferences, in the form of key/value pairs (akin to a `Map`), that will hang around between invocations of an activity. As the name suggests, the primary purpose is for you to store user-specified configuration details, such as the last feed the user looked at in your feed reader, or what sort order to use by default on a list, or whatever. Of course, you can store in the preferences whatever you like, so long as it is keyed by a `String` and has a primitive value (`boolean`, `String`, etc.)

Preferences can either be for a single activity or shared among all activities in an application. Other components, such as services, also can work with shared preferences.

Getting What You Want

To get access to the preferences, you have three APIs to choose from:

- `getPreferences()` from within your `Activity`, to access activity-specific preferences
- `getSharedPreferences()` from within your `Activity` (or other application `Context`), to access application-level preferences
- `getDefaultSharedPreferences()`, on `PreferenceManager`, to get the shared preferences that work in concert with Android's overall preference framework

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The first two take a security mode parameter. The right answer here is `MODE_PRIVATE`, so no other applications can access the file. The `getSharedPreferences()` method also takes a name of a set of preferences; `getPreferences()` effectively calls `getSharedPreferences()` with the activity's class name as the preference set name. The `getDefaultSharedPreferences()` method takes the `Context` for the preferences (e.g., your `Activity`).

All of those methods return an instance of `SharedPreferences`, which offers a series of getters to access named preferences, returning a suitably-typed result (e.g., `getBoolean()` to return a boolean preference). The getters also take a default value, which is returned if there is no preference set under the specified key.

Unless you have a good reason to do otherwise, you are best served using the third option above — `getDefaultSharedPreferences()` — as that will give you the `SharedPreferences` object that works with a `PreferenceActivity` by default, as will be described later in this chapter.

Stating Your Preference

Given the appropriate `SharedPreferences` object, you can use `edit()` to get an "editor" for the preferences. This object has a set of setters that mirror the getters on the parent `SharedPreferences` object. It also has:

1. `remove()` to get rid of a single named preference
2. `clear()` to get rid of all preferences
3. `apply()` or `commit()` to persist your changes made via the editor

The last one is important — if you modify preferences via the editor and fail to save the changes, those changes will evaporate once the editor goes out of scope. `commit()` is a blocking call, while `apply()` works asynchronously. Ideally, use `apply()` where possible, though it was only added in Android 2.3, so it may not be available to you if you are aiming to support earlier versions of Android than that.

Conversely, since the preferences object supports live changes, if one part of your application (say, an activity) modifies shared preferences, another part of your application (say, a service) will have access to the changed value immediately.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Introducing PreferenceActivity

You could roll your own activity to collect preferences from the user. On the whole, this is a bad idea. Instead, use preference XML resources and a `PreferenceActivity`.

Why?

One of the common complaints about Android developers is that they lack discipline, not following any standards or conventions inherent in the platform. For other operating systems, the device manufacturer might prevent you from distributing apps that violate their human interface guidelines. With Android, that is not the case — but this is not a blanket permission to do whatever you want. Where there is a standard or convention, please follow it, so that users will feel more comfortable with your app and their device.

Using a `PreferenceActivity` for collecting preferences is one such convention.

The linchpin to the preferences framework and `PreferenceActivity` is yet another set of XML data structures. You can describe your application's preferences in XML files stored in your project's `res/xml/` directory. Given that, Android can present a pleasant UI for manipulating those preferences, which are then stored in the `SharedPreferences` you get back from `getDefaultSharedPreferences()`.

To see how all of this works, take a look at the `Prefs/FragmentsBC` sample project.

What We Are Aiming For

This project's main activity hosts a `TableLayout`, into which we will load the values of five preferences:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/checkbox"/>

    <TextView
      android:id="@+id/checkbox"
      style="@style/value"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  </TableRow>

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/ringtone"/>

    <TextView
      android:id="@+id/ringtone"
      style="@style/value"/>
  </TableRow>

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/checkbox2"/>

    <TextView
      android:id="@+id/checkbox2"
      style="@style/value"/>
  </TableRow>

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/text"/>

    <TextView
      android:id="@+id/text"
      style="@style/value"/>
  </TableRow>

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/list"/>

    <TextView
      android:id="@+id/list"
      style="@style/value"/>
  </TableRow>

</TableLayout>
```

The above layout is used by `PreferenceContentsFragment`, which populates the right-hand column of `TextView` widgets at runtime in `onResume()`, pulling the values from the default `SharedPreferences` for our application:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.preffragsbc;

import android.app.Fragment;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class PreferenceContentsFragment extends Fragment {
  private TextView checkbox=null;
  private TextView ringtone=null;
  private TextView checkbox2=null;
  private TextView text=null;
  private TextView list=null;

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.content, parent, false);

    checkbox=(TextView)result.findViewById(R.id.checkbox);
    ringtone=(TextView)result.findViewById(R.id.ringtone);
    checkbox2=(TextView)result.findViewById(R.id.checkbox2);
    text=(TextView)result.findViewById(R.id.text);
    list=(TextView)result.findViewById(R.id.list);

    return(result);
  }

  @Override
  public void onResume() {
    super.onResume();

    SharedPreferences prefs=
        PreferenceManager.getDefaultSharedPreferences(getActivity());

    checkbox.setText(Boolean.valueOf(prefs.getBoolean("checkbox",
false)).toString());
    ringtone.setText(prefs.getString("ringtone", "<unset>"));
    checkbox2.setText(Boolean.valueOf(prefs.getBoolean("checkbox2",
false)).toString());
    text.setText(prefs.getString("text", "<unset>"));
    list.setText(prefs.getString("list", "<unset>"));
  }
}
```

The main activity, `FragmentsDemo`, simply loads `res/layout/main.xml`, which contains a `<fragment>` element pointing at `PreferenceContentsFragment`. It also defines an options menu, which we will examine later in this section.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The result is an activity showing the default values of the preferences when it is first run, since we have not set any values yet:

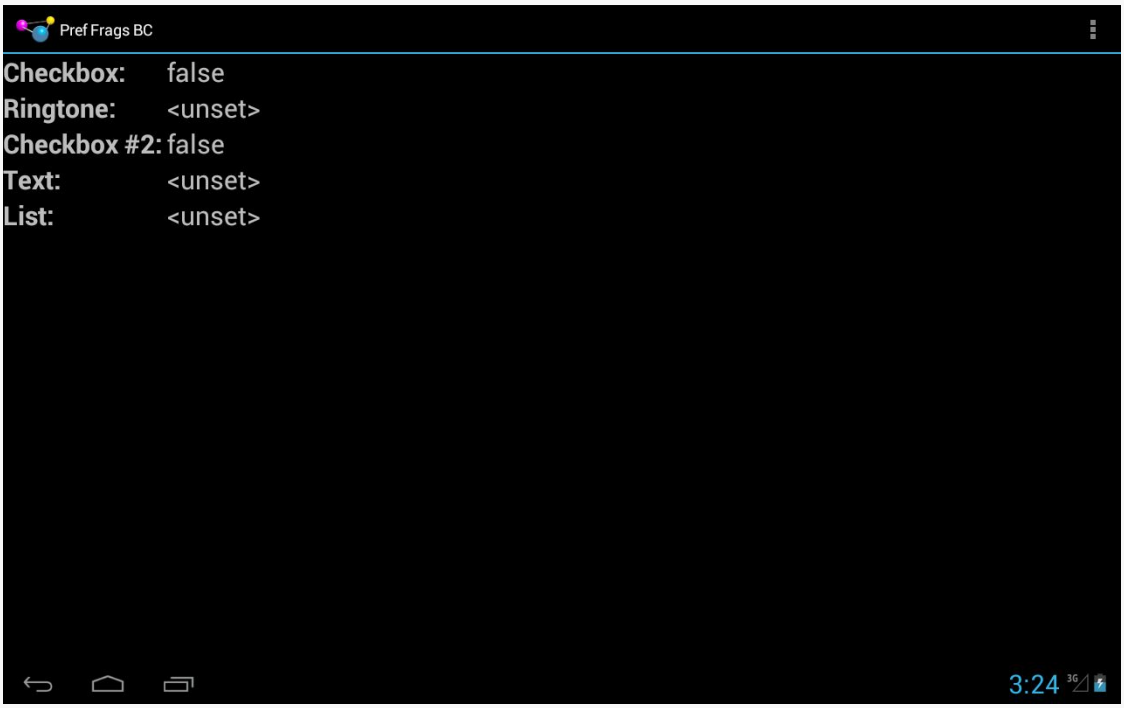


Figure 262: Activity Showing Preference Values

We will also have two flavors of a `PreferenceActivity`, to collect the preferences from the user. Those preferences will be divided into two "preference headers", following the two-pane preference UI adopted with Android 3.0:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

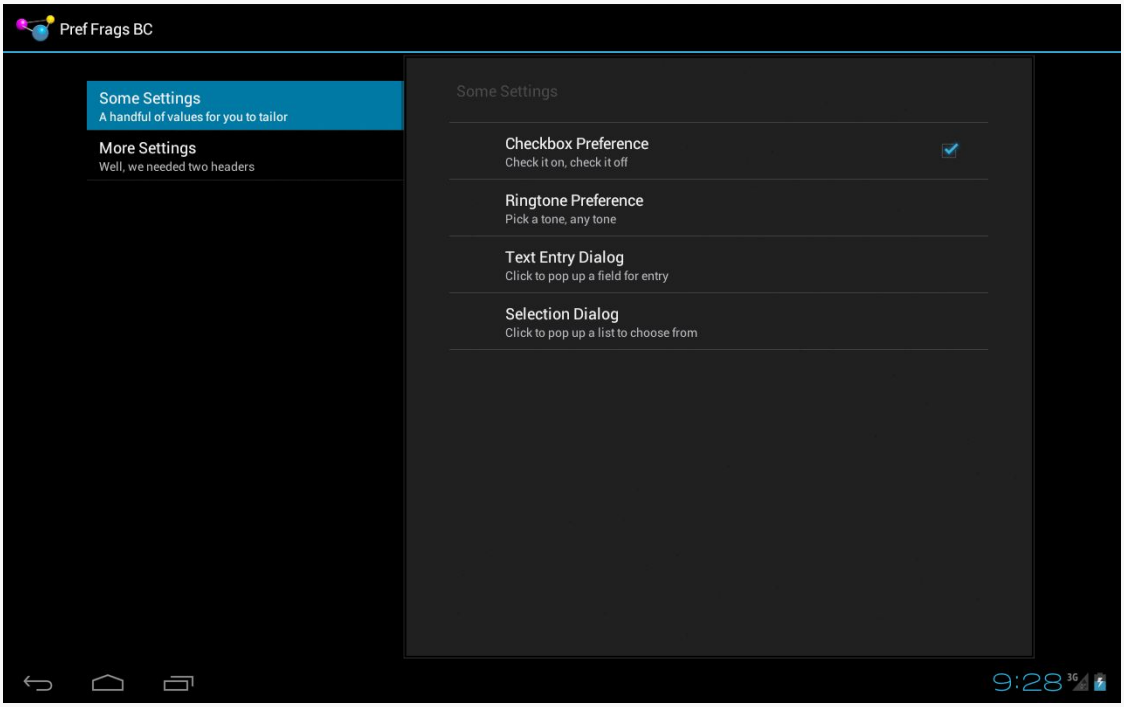


Figure 263: Android 4.0 PreferenceActivity, on Tablet

On a phone-sized screen, those panes become two separate screens, the first showing the list of headers:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

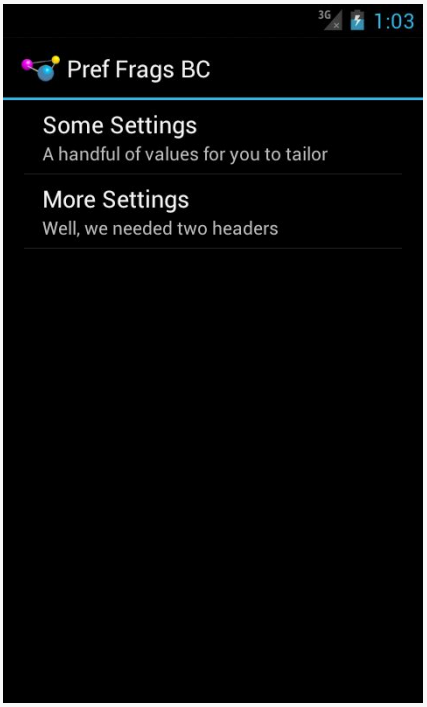


Figure 264: Android 4.0 PreferenceActivity, on Phone

and the second showing the contents of a specific header:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

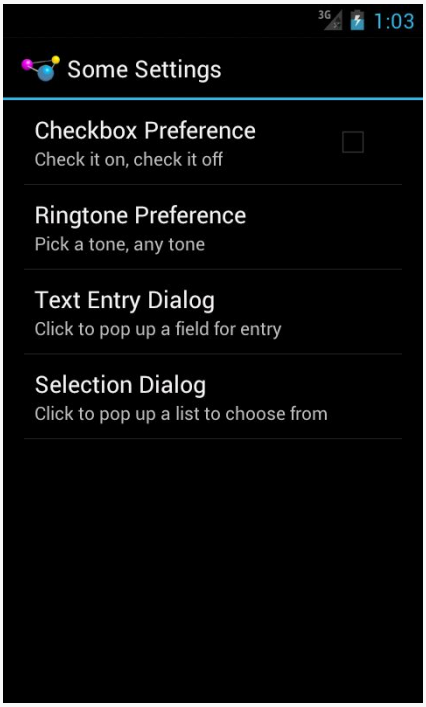


Figure 265: Android 4.0 PreferenceActivity, on Phone, Showing Second Preference Header

Defining Your Preferences

First, you need to tell Android what preferences you are trying to collect from the user.

To do this, you will need to add a `res/xml/` directory to your project, if one does not already exist. Then, for each preference header, you will want an XML file in `res/xml/` to contain the definition of the preferences you want to appear in that header.

The root element of this XML file will be `<PreferenceScreen>`, and it will contain child elements, one per preference.

For example, here is the second preference header's preferences, from `res/xml/preference2.xml`:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">

  <CheckBoxPreference
    android:key="checkbox2"
    android:summary="@string/pref5summary"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:title="@string/pref5title"/>

</PreferenceScreen>
```

There is a single `<CheckBoxPreference>` element inside the `<PreferenceScreen>`, allowing the user to toggle a boolean value via a `CheckBox` widget.

Each preference element has two attributes at minimum:

1. `android:key`, which is the key you use to look up the value in the `SharedPreferences` object via methods like `getInt()`
2. `android:title`, which is a few words identifying this preference to the user

You may also wish to consider having `android:summary`, which is a short sentence explaining what the user is to supply for this preference

We will examine more preference elements later in this chapter.

Defining Your Preference Headers

There is another XML resource you will need to define, one containing details about your preference headers. In this sample project, that is found in `res/xml/preference_headers.xml`:

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">

  <header
    android:fragment="com.commonsware.android.preffragsbc.EditPreferences$First"
    android:summary="@string/header1summary"
    android:title="@string/header1title">
  </header>
  <header
    android:fragment="com.commonsware.android.preffragsbc.EditPreferences$Second"
    android:summary="@string/header2summary"
    android:title="@string/header2title">
  </header>

</preference-headers>
```

Here, your root element is `<preference-headers>`, containing a series of `<header>` elements. Each `<header>` contains at least three attributes:

1. `android:fragment`, which identifies the Java class implementing the `PreferenceFragment` to use for this header, as is described in the next section

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. `android:title`, which is a few words identifying this header to the user

Once again, you may wish to also include `android:summary`, which is a short sentence explaining what the user will find inside of this header.

You can, if you wish, include one or more `<extra>` child elements inside the `<header>` element. These values will be put into the "arguments" `Bundle` that your `PreferenceFragment` can retrieve via `getArguments()`.

Creating Your PreferenceFragments

Preference XML, on API Level 11 and higher, is loaded by an implementation of `PreferenceFragment`. The mission of `PreferenceFragment` is to call `addPreferencesFromResource()` in `onCreate()`, supplying the resource ID of the preference XML to load for a particular preference header (e.g., `R.xml.preference2`).

There are two ways you can go about doing this. One is to create a dedicated `PreferenceFragment` subclass per preference header. The other is to create a single reusable `PreferenceFragment` implementation that can load up the preference XML for any preference header.

This sample app takes the former approach. Our `PreferenceActivity` implementation — named `EditPreferences` — has a pair of static inner classes representing the preference fragments:

```
public static class First extends PreferenceFragment {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    addPreferencesFromResource(R.xml.preferences);
  }
}

public static class Second extends PreferenceFragment {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    addPreferencesFromResource(R.xml.preferences2);
  }
}
```

Each of those `PreferenceFragment` implementations has an `onCreate()` method that calls `addPreferencesFromResource()` for its specific preferences.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We will illustrate the other approach, of creating a reusable `PreferenceFragment`, later in this chapter.

Creating Your PreferenceActivity

Our `PreferenceActivity` subclass is named `EditPreferences`, which contains little more than the two fragments shown above:

```
package com.commonsware.android.preffragsbc;

import android.os.Bundle;
import android.preference.PreferenceActivity;
import android.preference.PreferenceFragment;
import java.util.List;

public class EditPreferences extends PreferenceActivity {
  @Override
  public void onBuildHeaders(List<Header> target) {
    loadHeadersFromResource(R.xml.preference_headers, target);
  }

  @Override
  protected boolean isValidFragment(String fragmentName) {
    if (First.class.getName().equals(fragmentName)
        || Second.class.getName().equals(fragmentName)) {
      return(true);
    }

    return(false);
  }

  public static class First extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      addPreferencesFromResource(R.xml.preferences);
    }
  }

  public static class Second extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      addPreferencesFromResource(R.xml.preferences2);
    }
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`onBuildHeaders()` is where we supply the preference headers, via a call to `loadHeadersFromResource()`.

We also need to have an `isValidFragment()` method, that will return `true` if the supplied fragment name is one we should be showing in this `PreferenceActivity`, `false` otherwise. This will only be called on Android 4.4+. However, we need to set up the project build target (e.g., `compileSdkVersion` in Android Studio, Project > Preferences > Android in Eclipse) to API Level 19 or higher. Failing to have this method will cause your app to crash on Android 4.4+ devices, when the user tries to bring up one of your `PreferenceFragments`.

Types of Preferences

There are a variety of subclasses of `Preference` in the Android SDK for use with `PreferenceActivity`. This section will outline the major ones as of Android 4.4. Later in the book we will examine how to create your own custom `Preference` classes.

CheckBoxPreference and SwitchPreference

The sample application shown above has a pair of `CheckBoxPreference` elements, one per preference XML file. A `CheckBoxPreference` is an "inline" preference, in that the widget the user interacts with (in this case, a `CheckBox`) is part of the preference screen itself, rather than contained in a separate dialog.

`SwitchPreference` is functionally equivalent to `CheckBoxPreference`, insofar as both collect `boolean` values from the user. The difference is that `SwitchPreference` uses a `Switch` widget that the user slides left and right to toggle between "on" and "off" states. Also note that `SwitchPreference` was added in API Level 14 and therefore will not be available to older Android versions.

EditTextPreference

`EditTextPreference`, when tapped by the user, pops up a dialog that contains an `EditText` widget. You can configure this widget via attributes on the `<EditTextPreference>` element — in addition to standard preference attributes like `android:key`, you can include any attribute understood by `EditText`, such as `android:inputType`.

The value stored in the `SharedPreferences` is a string.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

RingtonePreference

`RingtonePreference` pops up a dialog with a list of ringtones installed on the device or emulator. However, note that the Android emulator does not come with any ringtones at the present time.

In addition to the standard preference attributes, you can include `android:showDefault`, indicating that the list should contain a "Default ringtone" option. If the user chooses this ringtone, they are effectively choosing the same ringtone that they have set up for incoming phone calls.

You can also use `android:showSilent`, which allows the user to choose a "Silence" pseudo-ringtone, to indicate not to play any ringtone.

For example, `res/xml/preferences.xml` from the sample project contains a `RingtonePreference`:

```
  <RingtonePreference
    android:key="ringtone"
    android:showDefault="true"
    android:showSilent="true"
    android:summary="@string/pref2summary"
    android:title="@string/pref2title"/>
```

The value stored in the `SharedPreferences` is a string, specifically the string representation of a `Uri` pointing to a `ContentProvider` that can serve up the ringtone for playback. The use of `ContentProvider` will be covered in a later chapter, and playing back media like ringtones will be covered in another later chapter.

ListPreference and MultiSelectListPreference

Visually, a `ListPreference` looks just like `RingtonePreference`, except that *you* control what goes into the list. You do this by specifying a pair of `string-array` resources in your preference XML.

String resources hold individual strings; string array resources hold a collection of strings. Typically, you will find string array resources in `res/values/arrays.xml` and related resource sets for translation. The `<string-array>` element has the `name` attribute to identify the resource, along with child `<item>` elements for the individual strings in the array.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For example, the sample application profiled in this chapter has a pair of string array resources in `res/values/arrays.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="cities">
    <item>Philadelphia</item>
    <item>Pittsburgh</item>
    <item>Allentown/Bethlehem</item>
    <item>Erie</item>
    <item>Reading</item>
    <item>Scranton</item>
    <item>Lancaster</item>
    <item>Altoona</item>
    <item>Harrisburg</item>
  </string-array>
  <string-array name="airport_codes">
    <item>PHL</item>
    <item>PIT</item>
    <item>ABE</item>
    <item>ERI</item>
    <item>RDG</item>
    <item>AVP</item>
    <item>LNS</item>
    <item>AOO</item>
    <item>MDT</item>
  </string-array>
</resources>
```

One of these (`cities`) will be the values the user sees in the list, and is associated with our preference via the `android:entries` attribute. The other (`airport_codes`) will be the *corresponding* values stored in the `SharedPreferences` as a string, and is associated with our preference via the `android:entryValues` attribute:

```
  <ListPreference
    android:dialogTitle="@string/listdialogtitle"
    android:entries="@array/cities"
    android:entryValues="@array/airport_codes"
    android:key="list"
    android:summary="@string/pref4summary"
    android:title="@string/pref4title"/>
```

We also use `android:dialogTitle` to provide the caption for the dialog:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

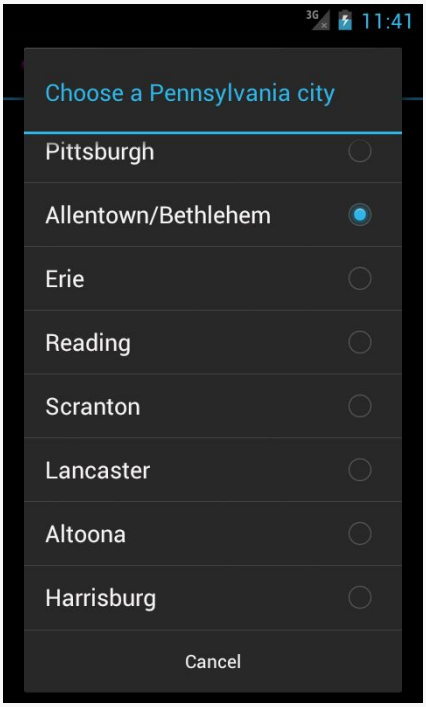


Figure 266: ListPreference on Android 4.0.3

When the user chooses a value (e.g., "Allentown/Bethlehem"), the corresponding value out of the other string array resource is stored in the `SharedPreferences` (e.g., "ABE").

`MultiSelectListPreference` works much the same way, except:

- The list contains checkboxes, not radio buttons
- The user can check multiple items
- The result is stored in a "string set" in the `SharedPreferences`, retrieved via `getStringSet()`
- It is only available on API Level 11 and higher

Intents for Headers or Preferences

If you have the need to collect some preferences that are beyond what the standard preferences can handle, you have some choices.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

One is to create a custom `Preference`. Extending `DialogPreference` to create your own `Preference` implementation is not especially hard. However, it does constrain you to something that can fit in a dialog.

Another option is to specify an `<intent>` element as a child of a `<header>` element. When the user taps on this header, your specified `Intent` is used with `startActivity()`, giving you a gateway to your own activity for collecting things that are beyond what the preference UI can handle. For example, you could have the following `<header>`:

```
<header android:icon="@drawable/something"
        android:title="Fancy Stuff"
        android:summary="Click here to transcend your
plane of existence">
  <intent android:action="com.commonsware.android.MY_CUSTOM_ACTION" />
</header>
```

Then, so long as you have an activity with an `<intent-filter>` specifying your desired action (`com.commonsware.android.MY_CUSTOM_ACTION`), that activity will get control when the user taps on the associated header.

Conditional Headers

The two-tier, headers-and-preferences approach is fine and helps to organize large rosters of preferences. However, it does tend to steer developers in the direction of displaying headers *all of the time*. For many apps, that is rather pointless, because there are too few preferences to collect to warrant having more than one header.

One alternative approach is to use the headers on larger devices, but skip them on smaller devices. That way, the user does not have to tap past a single-item `ListFragment` just to get to the actual preferences to adjust.

This is a wee bit tricky to implement. However, you have two options for how to accomplish it.

(The author would like to thank Richard Le Mesurier, whose question on this topic spurred the development of this section and its samples)

Option #1: Do Not Define the Headers

The basic plan in the first approach is to have smarts in `onBuildHeaders()` to handle this. `onBuildHeaders()` is the callback that Android invokes on our

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`PreferenceActivity` to let us define the headers to use in the master-detail pattern. If we want to have headers, we would supply them here; if we want to skip the headers, we would instead fall back to the classic (and, admittedly, deprecated) `addPreferencesFromResource()` method to load up some preference XML.

There is an `isMultiPane()` method on `PreferenceActivity`, starting with API Level 11, that will tell you if the activity will render with two fragments (master+detail) or not. In principle, this would be ideal to use. Unfortunately, it does not seem to be designed to be called from `onBuildHeaders()`. Similarly, `addPreferencesFromResource()` does not seem to be callable from `onBuildHeaders()`. Both are due to timing: `onBuildHeaders()` is called in the middle of the `PreferenceActivity` `onCreate()` processing.

So, we have to do some fancy footwork.

By examining the source code to `PreferenceActivity`, you will see that the logic that drives the single-pane vs. dual-pane UI decision boils down to:

```
onIsHidingHeaders() || !onIsMultiPane()
```

If that expression returns `true`, we are in single-pane mode; otherwise, we are in dual-pane mode. `onIsHidingHeaders()` will normally return `false`, while `onIsMultiPane()` will return either `true` or `false` based upon screen size.

So, we can leverage this information in a `PreferenceActivity` to conditionally load our headers, as seen in the `EditPreferences` class in the `Prefs/SingleHeader` sample project:

```
package com.commonsware.android.pref1header;

import android.os.Bundle;
import android.preference.PreferenceActivity;
import java.util.List;

public class EditPreferences extends PreferenceActivity {
  private boolean needResource=false;

  @SuppressWarnings("deprecation")
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (needResource) {
      addPreferencesFromResource(R.xml.preferences);
    }
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void onBuildHeaders(List<Header> target) {
    if (onIsHidingHeaders() || !onIsMultiPane()) {
      needResource=true;
    }
    else {
      loadHeadersFromResource(R.xml.preference_headers, target);
    }
  }

  @Override
  protected boolean isValidFragment(String fragmentName) {
    return(StockPreferenceFragment.class.getName().equals(fragmentName));
  }
}
```

Here, if we are in dual-pane mode, `onBuildHeaders()` populates the headers as normal. If, though, we are in single-pane mode, we skip that step and make note that we need to do some more work in `onCreate()`.

Then, in `onCreate()`, if we did not load our headers we use the classic `addPreferencesFromResource()` method.

The net result is that on Android 3.0+ tablets, we get the dual-pane, master-detail look with our one header, but on smaller devices (regardless of version), we roll straight to the preferences themselves.

Note that this sample application uses a single `PreferenceFragment` implementation, named `StockPreferenceFragment`:

```
package com.commonsware.android.pref1header;

import android.os.Bundle;
import android.preference.PreferenceFragment;

public class StockPreferenceFragment extends PreferenceFragment {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    int res=
        getActivity().getResources()
                     .getIdentifier(getArguments().getString("resource"),
                                    "xml",
                                    getActivity().getPackageName());

    addPreferencesFromResource(res);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`StockPreferenceFragment` does what it is supposed to: call `addPreferencesFromResource()` in `onCreate()` with the resource ID of the preferences to load. However, rather than hard-coding a resource ID, as we normally would, we look it up at runtime.

The `<extra>` elements in our preference header XML supply the name of the preference XML to be loaded:

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">

  <header

android:fragment="com.commonsware.android.pref1header.StockPreferenceFragment"
    android:summary="@string/header1summary"
    android:title="@string/header1title">
    <extra
      android:name="resource"
      android:value="preferences"/>
  </header>

</preference-headers>
```

We get that name via the arguments `Bundle` (`getArguments().getString("resource")`).

To look up a resource ID at runtime, we can use the `Resources` object, available from our activity via a call to `getResources()`. `Resources` has a method, `getIdentifier()`, that will return a resource ID given three pieces of information:

1. The base name of the resource (in our case, the value retrieved from the `<extra>` element)
2. The type of the resource (e.g., `"xml"`)
3. The package holding the resource (in our case, our own package, retrieved from our activity via `getPackageName()`)

Note that `getIdentifier()` uses reflection to find this value, and so there is some overhead in the process. Do not use `getIdentifier()` in a long loop – cache the value instead.

The net is that `StockPreferenceFragment` loads the preference XML described in the `<extra>` element, so we do not need to create separate `PreferenceFragment` implementations per preference header.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Option #2: Go Directly to the Fragment

The advantage of the above approach is that it works with Android's own logic of whether to display the master-detail fragments or just one at a time. However, that logic — the fact that `onIsHidingHeaders() || !onIsMultiPane()` determines the look of the activity — is not documented, and therefore may change in future Android releases.

Another option is to launch your `PreferenceActivity` in such a way that tells Android to skip showing the headers. This approach is better documented and therefore perhaps more stable. This can also be used in cases where you do want headers sometimes, but at other times you want to route the user to a specific `PreferenceFragment`. The downside is that this technique only works on API Level 11+.

To see how this works, take a look at the `Prefs/SingleHeader2` sample project.

Our `EditPreferences` class is the same implementation as in the original sample for this chapter, except that we only load up the single XML resource's worth of preferences:

```
package com.commonsware.android.pref1header;

import android.preference.PreferenceActivity;
import java.util.List;

public class EditPreferences extends PreferenceActivity {
  @Override
  public void onBuildHeaders(List<Header> target) {
    loadHeadersFromResource(R.xml.preference_headers, target);
  }

  @Override
  protected boolean isValidFragment(String fragmentName) {
    return(StockPreferenceFragment.class.getName().equals(fragmentName));
  }
}
```

However, there is a change in our main activity (`FragmentsDemo`). Before, when the user chose the "Settings" action bar overflow item, we would just call `startActivity()` to bring up `EditPreferences`. Now, we delegate that work to an `editPrefs()` method on `FragmentsDemo`, which will have the smarts to control *how* we bring up the `EditPreferences` activity:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
private void editPrefs() {
  Intent i=new Intent(this, EditPreferences.class);

  i.putExtra(PreferenceActivity.EXTRA_SHOW_FRAGMENT,
             StockPreferenceFragment.class.getName());

  Bundle b=new Bundle();

  b.putString("resource", "preferences");

  i.putExtra(PreferenceActivity.EXTRA_SHOW_FRAGMENT_ARGUMENTS, b);

  startActivity(i);
}
```

Here, we will add two extras to our `Intent`:

- `EXTRA_SHOW_FRAGMENT`, set to the fully-qualified class name of the `PreferenceFragment` to be displayed, here obtained by calling `getName()` on the `Class` object for `StockPreferenceFragment`
- `EXTRA_SHOW_FRAGMENT_ARGUMENTS`, set to a `Bundle` containing the same values that would ordinarily be loaded from the `<extra>` elements in the preference header XML resource (in our case, the name of the preference XML resource to load)

Those extras will be automatically handled by `PreferenceActivity` (on API Level 11+) and will have the effect of directly taking the user to our one-and-only fragment, bypassing the headers.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #13 - Using Some Preferences

Now that we have the core reading functionality working, we can start to add other features for the user.

One common thing in Android applications is to collect preferences from the user, tailoring the way the app behaves. In the case of `EmPubLite`, we will initially track two preferences:

- Whether the user wants to return to the book on the same chapter (page in the `ViewPager`) that they were on when they last were reading the book
- Whether the user wants us to keep the screen on, so they do not have to keep tapping the screen to prevent Android's automatic sleep mode from kicking in

In this tutorial, we will collect and use these two preferences.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Defining the Preference XML Files

We need two XML files to define what preferences we wish to collect. One will define the preference headers (the left column of the two-pane tablet preference UI).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The other will define the preferences that we wish to collect for the one header we will define.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Add six new `<string>` elements to `res/values/strings.xml`:

```
<string name="prefdesc">Settings for use of EmPubLite</string>
<string name="preftitle">Display and Navigation</string>
<string name="lastpositiontitle">Save Last Position</string>
<string name="lastpositionsummary">Save the last chapter you were viewing and
open up on that chapter when re-opening the app</string>
<string name="keepscreenon_summary">Keep the screen powered on while the reader
is in the foreground</string>
<string name="keepscreenon_title">Keep Screen On</string>
```

Right click over `res/` in your project, and choose New > "Android resource directory" from the context menu. Change the "Resource type" drop-down to be "xml", then click OK to create the directory.

Then, right-click over your new `res/xml/` directory and choose New > "XML resource file" from the context menu. Fill in `preference_headers.xml` in the "New XML Resource File" dialog, then click OK to create the file. It will open up into an XML editor, into which you can paste the following content:

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">

  <header
    android:fragment="com.commonsware.empublite.Preferences$Display"
    android:summary="@string/prefdesc"
    android:title="@string/preftitle">
    <extra
      android:name="resource"
      android:value="pref_display"/>
  </header>

</preference-headers>
```

Repeat the above steps to create a `res/xml/pref_display.xml` file, with the following content:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:empub="http://schemas.android.com/apk/res-auto">
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <CheckBoxPreference
    android:defaultValue="false"
    android:key="saveLastPosition"
    android:summary="@string/lastpositionsummary"
    android:title="@string/lastpositiontitle"/>
  <CheckBoxPreference
    android:defaultValue="false"
    android:key="keepScreenOn"
    android:summary="@string/keepscreenon_summary"
    android:title="@string/keepscreenon_title"/>

</PreferenceScreen>
```

Eclipse

Double-click on the `res/values/strings.xml` file in your Package Explorer. Use the "Add..." button to define a new string resource, with a name of `prefdesc` and a value of `Settings for use of EmPubLite`. Then, use the "Add..." button again to define another string resource, with a name of `preftitle` and a value of `Display and Navigation`. Repeat the process with four more string resources:

- `lastpositionsummary` = `Save the last chapter you were viewing and open up on that chapter when re-opening the app`
- `lastpositiontitle` = `Save Last Position`
- `keepscreenon_summary` = `Keep the screen powered on while the reader is in the foreground`
- `keepscreenon_title` = `Keep Screen On`

Right-click over the `res/` folder, and choose New > Folder from the context menu. Fill in `xml` as the folder name, then click "Finish" to create the folder.

Right-click over the `xml/` folder, and choose New > File from the context menu. Fill in `preference_headers.xml` as the name and `preference-headers` as the root element, then click "Finish" to create the file. Switch to the `preference_headers.xml` sub-tab of the newly-opened editor and paste in the following:

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">

  <header
    android:fragment="com.commonsware.empublite.Preferences$Display"
    android:summary="@string/prefdesc"
    android:title="@string/preftitle">
    <extra
      android:name="resource"
      android:value="pref_display"/>
  </header>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
</preference-headers>
```

You will get an error from Eclipse complaining that `com.commonsware.empublite.Preferences$Display` is not recognized — we will correct that problem later in this tutorial.

Then, right-click over the `xml` folder, and choose New > File from the context menu. Fill in `pref_display.xml` as the name, leaving the root element alone, then click "Finish" to create the file. Switch to the `pref_display.xml` sub-tab of the newly-opened editor and paste in the following:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:empub="http://schemas.android.com/apk/res-auto">

  <CheckBoxPreference
    android:defaultValue="false"
    android:key="saveLastPosition"
    android:summary="@string/lastpositionsummary"
    android:title="@string/lastpositiontitle"/>
  <CheckBoxPreference
    android:defaultValue="false"
    android:key="keepScreenOn"
    android:summary="@string/keepscreenon_summary"
    android:title="@string/keepscreenon_title"/>

</PreferenceScreen>
```

Note that while the code listing may show the root element wrapping onto a second line, it really should be all on one line.

Step #2: Creating Our PreferenceActivity

We now need an implementation of `PreferenceActivity`, named `Preferences`, to load our preference XML, using a static inner class implementation of a `PreferenceFragment`, named `Display`.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `Preferences` as the name and click OK to create the empty class.

In the `Preferences` class that is created, replace the current implementation with the following:

```
package com.commonsware.empublite;

import android.os.Bundle;
import android.preference.PreferenceActivity;
import android.preference.PreferenceFragment;
import java.util.List;

public class Preferences extends PreferenceActivity {
  @Override
  public void onBuildHeaders(List<Header> target) {
    loadHeadersFromResource(R.xml.preference_headers, target);
  }

  @Override
  protected boolean isValidFragment(String fragmentName) {
    if (Display.class.getName().equals(fragmentName)) {
      return(true);
    }

    return(false);
  }

  public static class Display extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      addPreferencesFromResource(R.xml.pref_display);
    }
  }
}
```

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Then, open up `AndroidManifest.xml` and add the following `<activity>` element inside the `<application>` element:

```
<activity android:name="Preferences"></activity>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Eclipse

Right-click over the EmPubLite project in the Package Explorer, then choose New > Other from the context menu. In the list of possible things to add to your project, choose "Android Activity" and click Next. In the list of templates, choose "Empty Activity" and click Next.

In the last page of the new-activity wizard, fill in:

- `Preferences` in the "Activity Name" field
- `pleasedeleteme` for the "Layout Name"
- `pleasedeletemetoo` for the "Title"

Leave the rest of the wizard at its defaults, then click on Finish.

In the `Preferences` class that is created, replace the current implementation with the source shown in the Android Studio instructions above.

You can also delete the `/res/layout/pleasedeleteme.xml` file, by right-clicking over it and choosing "Delete..." from the context menu.

Finally, open up `res/values/strings.xml` and delete the `title_activity_simple_content` that the wizard added, which we are no longer using. At this point, you can save your changes to this file (e.g., `<Ctrl>-<S>`).

Step #3: Adding To Our Action Bar

Of course, having this activity does us no good if we cannot start it up, so we need to add another hook to our action bar configuration for that.

First, copy the `13_extra_actions_settings` icons from the `Action Bar Icons/holo_dark/` directory from the "Action Bar Icon Pack" into your project. In the `13_extra_actions_settings` directory, you will see versions of this icon in various densities. Copy those icon files into their corresponding directories in your project. You can do this by dragging the file from your development desktop OS into the project explorer window and dropping it into the proper directory.

Add the following XML element to `res/menu/options.xml` as the first child of the `<menu>` root element:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    <item
        android:id="@+id/settings"
        android:icon="@drawable/ic_action_settings"
        android:showAsAction="never"
        android:title="@string/settings">
    </item>
```

You will also need to add a `settings` string resource, with a value of `Settings`:

```
<string name="settings">Settings</string>
```

Step #4: Launching the PreferenceActivity

The only thing yet needed to allow the user to get to the preferences is to add another `case` to the `switch()` statement in `onOptionsItemSelected()` of `EmPubLiteActivity`:

```
      case R.id.settings:
        startActivity(new Intent(this, Preferences.class));

        return(true);
```

Now, if you run this in an emulator or device, you will see the new option in the action bar overflow:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

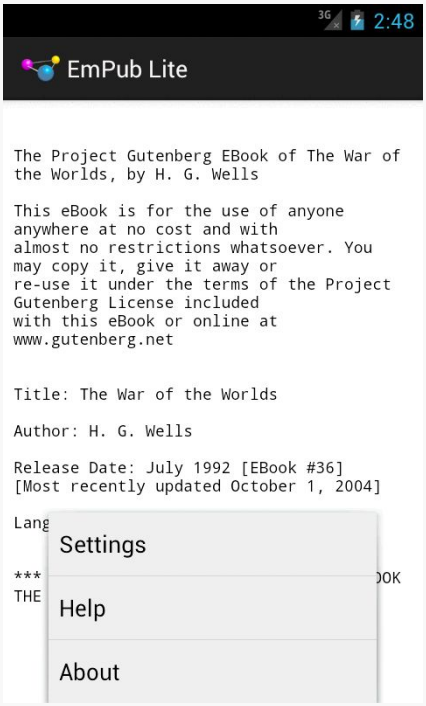


Figure 267: EmPubLite, With Revised Action Bar

Choosing the "Settings" option brings up the list of preference headings:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 268: Our Preference Headings

Tapping on the "Display and Navigation" heading brings up our two preferences:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

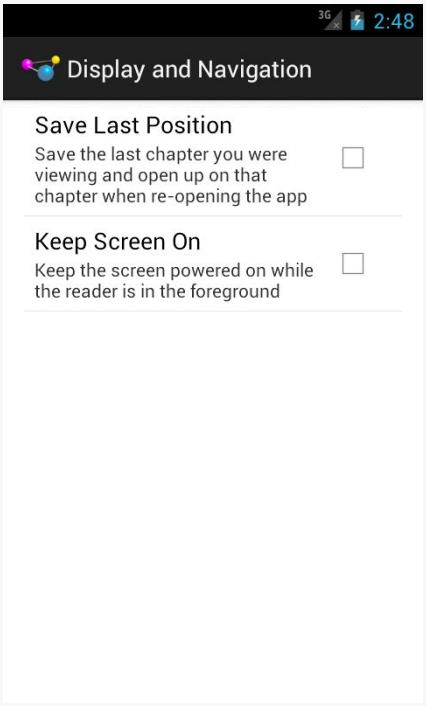


Figure 269: Our Preferences

On a tablet, we see the headings and the selected headings' worth of preferences at the same time:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

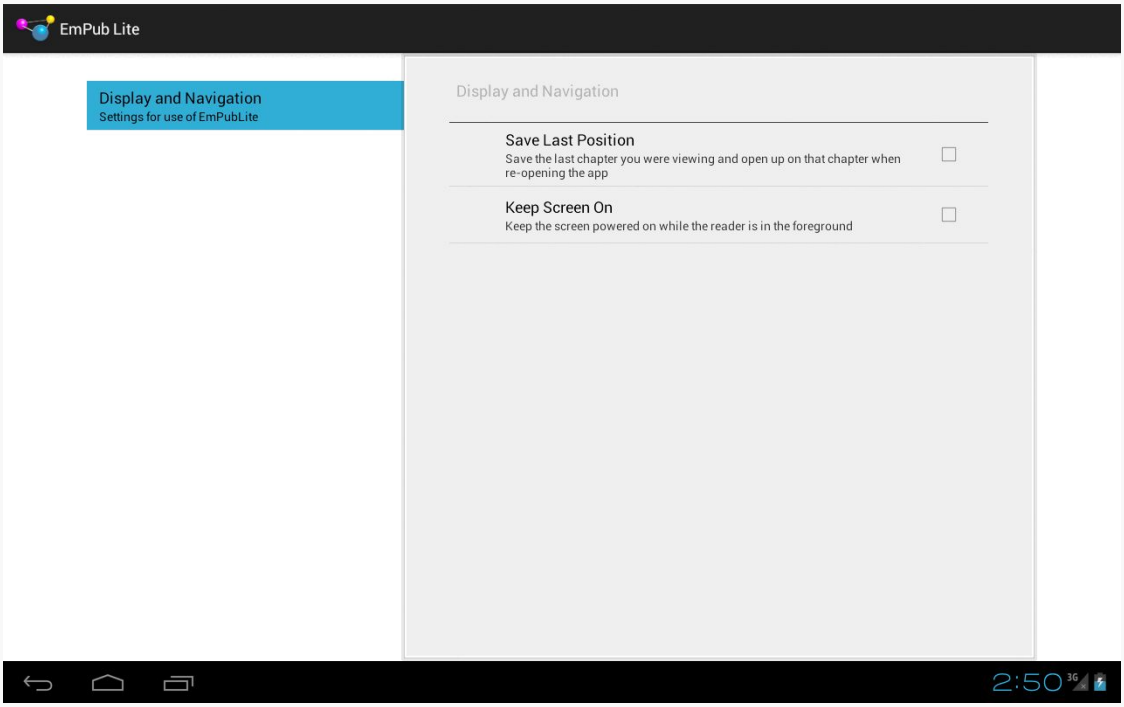


Figure 270: The Entire PreferenceActivity, On a Tablet

Step #5: Loading the Preferences

Now, we need to actually arrange to load the preferences on a background thread. As noted, this will be handled by our `ModelFragment`, much as it handles the loading of the book contents.

First, add a private data member named `prefs`, of type `SharedPreferences`, to `ModelFragment`:

```
  private SharedPreferences prefs=null;
```

Then, add a `getPrefs()` method to `ModelFragment` that returns the `prefs` value:

```
  public SharedPreferences getPrefs() {
    return(prefs);
  }
```

Next, revise `LoadThread` to:

- Replace the `assets` data member with a `ctxt` data member of type `Context`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Take in a `Context` in the constructor, instead of an `AssetManager` (this way, even if for some strange reason our original activity is destroyed and recreated while we are loading the preferences, we will not be leaking the original activity)
- Save the application context (from `getApplicationContext()` on `Context`) in a data member, instead of an `AssetManager`
- Call `getAssets()` on that `Context` in `run()`, instead of using the former `AssetManager`
- Also retrieve the `SharedPreferences` in `run()`

```
private class LoadThread extends Thread {
  private Context ctxt=null;

  LoadThread(Context ctxt) {
    super();

    this.ctxt=ctxt.getApplicationContext();
    Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
  }

  @Override
  public void run() {
    prefs=PreferenceManager.getDefaultSharedPreferences(ctxt);

    Gson gson=new Gson();

    try {
      InputStream is=ctxt.getAssets().open("book/contents.json");
      BufferedReader reader=
          new BufferedReader(new InputStreamReader(is));

      contents=gson.fromJson(reader, BookContents.class);
      EventBus.getDefault().post(new BookLoadedEvent(contents));
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
    }
  }
}
```

This has our `LoadThread` load both the `SharedPreferences` and the `BookContents`, and do so in a known order (`SharedPreferences` first).

You will need to modify `onAttach()` to just pass in the `Activity` to the `LoadThread` constructor:

```
@Override
public void onAttach(Activity host) {
  super.onAttach(host);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (contents == null) {
      new LoadThread(host).start();
    }
  }
```

The resulting `ModelFragment` should look like:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.Fragment;
import android.content.Context;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.os.Process;
import android.preference.PreferenceManager;
import android.util.Log;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import com.google.gson.Gson;
import de.greenrobot.event.EventBus;

public class ModelFragment extends Fragment {
  private BookContents contents=null;
  private SharedPreferences prefs=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setRetainInstance(true);
  }

  @Override
  public void onAttach(Activity host) {
    super.onAttach(host);

    if (contents == null) {
      new LoadThread(host).start();
    }
  }

  public BookContents getBook() {
    return(contents);
  }

  public SharedPreferences getPrefs() {
    return(prefs);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private class LoadThread extends Thread {
    private Context ctxt=null;

    LoadThread(Context ctxt) {
      super();

      this.ctxt=ctxt.getApplicationContext();
      Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
    }

    @Override
    public void run() {
      prefs=PreferenceManager.getDefaultSharedPreferences(ctxt);

      Gson gson=new Gson();

      try {
        InputStream is=ctxt.getAssets().open("book/contents.json");
        BufferedReader reader=
            new BufferedReader(new InputStreamReader(is));

        contents=gson.fromJson(reader, BookContents.class);
        EventBus.getDefault().post(new BookLoadedEvent(contents));
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
      }
    }
  }
}
```

Step #6: Saving the Last-Read Position

One preference is to restore our current page in the `ViewPager` when the user later re-opens the app. To make that work, we need to start saving the current page as the user leaves the app. And, we may as well use our freshly-minted `SharedPreferences` to store this value.

We need a key under which we will store this value in the `SharedPreferences`, so add a new static data member to `EmPubLiteActivity`:

```
private static final String PREF_LAST_POSITION="lastPosition";
```

We are also going to need access to our `ModelFragment` from outside of `onCreate()` in `EmPubLiteActivity`. Add a `ModelFragment` data member named `mfrag`:

```
  private ModelFragment mfrag=null;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Then, modify `onResume()` to refer to the `mfrag` data member, replacing the former `mfrag` local variable:

```
@Override
public void onResume() {
  super.onResume();

  EventBus.getDefault().register(this);

  if (adapter==null) {
    mfrag=
        (ModelFragment)getFragmentManager().findFragmentByTag(MODEL);

    if (mfrag == null) {
      mfrag=new ModelFragment();
      getFragmentManager().beginTransaction().add(mfrag, MODEL).commit();
    }
    else if (mfrag.getBook() != null) {
      setupPager(mfrag.getBook());
    }
  }
}
```

Next, update `onPause()` on `EmPubLiteActivity` to track the page of the `ViewPager` that the user is on at the point in time when `onPause()` is called:

```
@Override
public void onPause() {
  EventBus.getDefault().unregister(this);

  if (mfrag.getPrefs() != null) {
    int position=pager.getCurrentItem();

    mfrag.getPrefs().edit().putInt(PREF_LAST_POSITION, position)
        .apply();
  }

  super.onPause();
}
```

Here, we check to see that we have the `SharedPreferences` loaded — odds are that we do, but we cannot be certain. If we do have access to the `SharedPreferences`, we find out the current position within the `ViewPager` via `getCurrentItem()` (e.g., `0` for the first page). We then obtain a `SharedPreferences.Editor` and use it to save this position value in the `SharedPreferences`, keyed as `PREF_LAST_POSITION`, using `apply()` to persist the changes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Step #7: Restoring the Last-Read Position

Now that we are saving this position data, we can start to use it.

Our preference XML has our key to the "Save Last Position" preference, but we need it in Java code as well, so add another static data member to `EmPubLiteActivity`:

```
private static final String PREF_SAVE_LAST_POSITION="saveLastPosition";
```

Add the following lines to the end of `setupPager()` in `EmPubLiteActivity`:

```
    SharedPreferences prefs=mfrag.getPrefs();

    if (prefs != null) {
      if (prefs.getBoolean(PREF_SAVE_LAST_POSITION, false)) {
        pager.setCurrentItem(prefs.getInt(PREF_LAST_POSITION, 0));
      }
    }
```

Here, we check to see if the user has enabled having us restore the last-saved position (defaulting to `false`). If the user has, we retrieve the last-saved position (defaulting to `0`, or the first page), and call `setCurrentItem()` on the `ViewPager` to shift to that particular page.

If you run this in a device or emulator, check the "Save Last Position" preference checkbox, flip ahead a couple of chapters, exit the app via the BACK button, and go back into the app, you will see that you are taken back to the chapter you were last reading.

Step #9: Keeping the Screen On

Our other preference is whether or not the screen should stay on, without user input, while we are reading the book. The bare-bones implementation of this requires just two lines of additional code.

First, we need to define another static data member on `EmPubLiteActivity`, this time with the key for our keep-screen-on preference:

```
private static final String PREF_KEEP_SCREEN_ON="keepScreenOn";
```

Then, add one more line to `setupPager()` in `EmPubLiteActivity`, inside of the `if` block:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
pager.setKeepScreenOn(prefs.getBoolean(PREF_KEEP_SCREEN_ON, false));
```

This will give you:

```
  private void setupPager(BookContents contents) {
    adapter=new ContentsAdapter(this, contents);
    pager.setAdapter(adapter);
    findViewById(R.id.progressBar1).setVisibility(View.GONE);
    pager.setVisibility(View.VISIBLE);

    SharedPreferences prefs=mfrag.getPrefs();
    if (prefs != null) {
      if (prefs.getBoolean(PREF_SAVE_LAST_POSITION, false)) {
        pager.setCurrentItem(prefs.getInt(PREF_LAST_POSITION, 0));
      }

      pager.setKeepScreenOn(prefs.getBoolean(PREF_KEEP_SCREEN_ON, false));
    }
  }
```

`setKeepScreenOn()`, called on any `View`, will keep the screen lit and active without continuous user input, so long as that `View` is on the screen.

This approach is somewhat limited, in that we are only setting this during the call to `setupPager()`. If the user changes the preference value, that change would only take effect when the activity was restarted (e.g., user rotates the screen, user exits the app via BACK and returns later).

The simplest way for us to have this take more immediate effect is to realize that `EmPubLiteActivity` will be paused and stopped when the `Preferences` activity is on the screen, and will be started and resumed when the user is done adjusting preferences. So, we can simply augment `onResume()` to also update the screen-on setting:

```
  @Override
  public void onResume() {
    super.onResume();

    EventBus.getDefault().register(this);

    if (adapter==null) {
      mfrag=
          (ModelFragment)getFragmentManager().findFragmentByTag(MODEL);

      if (mfrag == null) {
        mfrag=new ModelFragment();
        getFragmentManager().beginTransaction().add(mfrag, MODEL).commit();
      }
      else if (mfrag.getBook() != null) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        setupPager(mfrag.getBook());
      }
    }

    if (mfrag.getPrefs() != null) {
      pager.setKeepScreenOn(mfrag.getPrefs()
          .getBoolean(PREF_KEEP_SCREEN_ON, false));
    }
  }
```

Of course, we may not *have* the `SharedPreferences` yet, when the app is first starting up, so we avoid making any changes in that case.

If you run this on a device (note: *not* an emulator), you can play with this preference and see the changes in the screen's behavior.

In Our Next Episode…

… we will allow the user to write, save, and delete notes for the currently-viewed chapter, using a database.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

SQLite Databases

Besides `SharedPreferences` and your own file structures, the third primary means of persisting data locally on Android is via SQLite. For many applications, SQLite is the app's backbone, whether it is used directly or via some third-party wrapper.

This chapter will focus on how you can directly work with SQLite to store relational data.

Introducing SQLite

SQLite is a very popular embedded database, as it combines a clean SQL interface with a very small memory footprint and decent speed. Moreover, it is public domain, so everyone can use it. Lots of firms (Adobe, Apple, Google, Symbian) and open source projects (Mozilla, PHP, Python) all ship products with SQLite.

For Android, SQLite is "baked into" the Android runtime, so every Android application can create SQLite databases. Since SQLite uses a SQL interface, it is fairly straightforward to use for people with experience in other SQL-based databases. However, its native API is not JDBC, and JDBC might be too much overhead for a memory-limited device like a phone, anyway. Hence, Android programmers have a different API to learn — the good news being is that it is not that difficult.

This chapter will cover the basics of SQLite use in the context of working on Android. It by no means is a thorough coverage of SQLite as a whole. If you want to learn more about SQLite, the SQLite Web site may help.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Thinking About Schemas

SQLite is a typical relational database, containing tables (themselves consisting of rows and columns), indexes, and so on. Your application will need its own set of tables and so forth for holding whatever data you wish to hold. This structure is generally referred to as a "schema".

It is likely that your schema will need to change over time. You might add new tables or columns in support of new features. Or, you might significantly reorganize your data structure and wind up dropping some tables while moving the data into new ones.

As a result, when you ship an update to your application to your users, not only will your Java code change, but the *expectations* of that Java code will change as well, with respect to what your database schema will look like. Version 1 of your app will use your original schema, but by the time you ship, say, version 5 of the app, you might need an adjusted schema.

Android has facilities to assist you with handling changing database schemas, mostly centered around the `SQLiteOpenHelper` class.

Start with a Helper

`SQLiteOpenHelper` is designed to consolidate your code related to two very common problems:

1. What happens the very first time when your app is run on a device after it is installed? At this point, we do not yet have a database, and so you will need to create your tables, indexes, starter data, and so on.
2. What happens the very first time when an upgraded version of your app is run on a device, where the upgraded version is expecting a newer database schema? Your database will still be on the old schema from the older edition of the app. You will need to have a chance to alter the database schema to match the needs of the rest of your app.

`SQLiteOpenHelper` wraps up the logic to create and upgrade a database, per your specifications, as needed by your application. You will need to create a custom subclass of `SQLiteOpenHelper`, implementing three methods at minimum:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. The constructor, chaining upward to the `SQLiteOpenHelper` constructor. This takes the `Context` (e.g., an `Activity`), the name of the database, an optional cursor factory (typically, just pass `null`), and an integer representing the version of the database schema you are using (typically start at `1` and increment from there).
2. `onCreate()`, called when there is no database and your app needs one, which passes you a `SQLiteDatabase` object, pointing at a newly-created database, that you use to populate with tables and initial data, as appropriate.
3. `onUpgrade()`, called when the schema version you are seeking does not match the schema version of the database, which passes you a `SQLiteDatabase` object and the old and new version numbers, so you can figure out how best to convert the database from the old schema to the new one.

To see how all this SQLite stuff works in practice, we will examine the Database/ConstantsROWID sample application. This application pulls a bunch of gravitational constants from the `SensorManager` class, puts them in a database table, displays them in a `ListFragment`, and allows the user to add new ones via the action bar.

First, we need a `SQLiteOpenHelper` subclass, here named `DatabaseHelper`.

The `DatabaseHelper` constructor chains to the superclass and supplies the name of the database (held in a `DATABASE_NAME` static data member) and the version number of our database schema (held in `SCHEMA`):

```
public class DatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="constants.db";
  private static final int SCHEMA=1;
  static final String TITLE="title";
  static final String VALUE="value";
  static final String TABLE="constants";

  public DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, SCHEMA);
  }
```

We also need an `onCreate()` method, which will be called and passed a `SQLiteDatabase` object when a database needs to be newly created. Below you will see the `DatabaseHelper` implementation of `onCreate()`, though we will get into how it is using the `SQLiteDatabase` object more later in this chapter:

```
  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE constants (title TEXT, value REAL);");
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    ContentValues cv=new ContentValues();

    cv.put(TITLE, "Gravity, Death Star I");
    cv.put(VALUE, SensorManager.GRAVITY_DEATH_STAR_I);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Earth");
    cv.put(VALUE, SensorManager.GRAVITY_EARTH);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Jupiter");
    cv.put(VALUE, SensorManager.GRAVITY_JUPITER);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Mars");
    cv.put(VALUE, SensorManager.GRAVITY_MARS);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Mercury");
    cv.put(VALUE, SensorManager.GRAVITY_MERCURY);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Moon");
    cv.put(VALUE, SensorManager.GRAVITY_MOON);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Neptune");
    cv.put(VALUE, SensorManager.GRAVITY_NEPTUNE);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Pluto");
    cv.put(VALUE, SensorManager.GRAVITY_PLUTO);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Saturn");
    cv.put(VALUE, SensorManager.GRAVITY_SATURN);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Sun");
    cv.put(VALUE, SensorManager.GRAVITY_SUN);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, The Island");
    cv.put(VALUE, SensorManager.GRAVITY_THE_ISLAND);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Uranus");
    cv.put(VALUE, SensorManager.GRAVITY_URANUS);
    db.insert(TABLE, TITLE, cv);

    cv.put(TITLE, "Gravity, Venus");
    cv.put(VALUE, SensorManager.GRAVITY_VENUS);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    db.insert(TABLE, TITLE, cv);
  }
```

Suffice it to say for the moment that it is creating a `constants` table and inserting several rows into it, all wrapped in a transaction.

We also need `onUpgrade()`... even though it should never be called right now:

```
  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion,
                        int newVersion) {
    throw new RuntimeException("How did we get here?");
  }
```

After all, right now, we only have one version of our schema (`1`) and therefore will have no need to upgrade. If, in the future, we change `SCHEMA` to a higher value (e.g., `2`), and we upgrade our app on a device that had previously been run with our earlier schema, *then* we will be called with `onUpgrade()`. We are passed the old and new schema versions, so we know what needs to be upgraded.

Bear in mind that users do not necessarily have to take on each of your application updates, and so you might find that a user skipped a schema version:

- You release an app on Monday, with schema version `1`
- A user installs your app on Tuesday and runs it, creating a database via `onCreate()`
- You release an upgraded app on Wednesday, with schema version `2`
- You release yet another upgrade on Thursday, with schema version `3`
- The user installs your upgrade, now needing a schema version `3` database instead of the version `1` presently on the device, triggering a call to `onUpgrade()`

There are two other methods you can elect to override in your `SQLiteOpenHelper`, if you feel the need:

- You can override `onOpen()`, to get control when somebody opens this database. Usually, this is not required.
- Android 3.0 introduced `onDowngrade()`, which will be called if the code requests an older schema than what is in the database presently. This is the converse of `onUpgrade()` — if your version numbers differ, one of these two methods will be invoked. Since normally you are moving forward with updates, you can usually skip `onDowngrade()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Employing Your Helper

To use your `SQLiteOpenHelper` subclass, create and hold onto an instance of it. Then, when you need a `SQLiteDatabase` object to do queries or data modifications, ask your `SQLiteOpenHelper` to `getReadableDatabase()` or `getWritableDatabase()`, depending upon whether or not you will be changing its contents.

For example, the `ConstantsFragment` from the sample app creates a `DatabaseHelper` instance in `onViewCreated()` and holds onto it in a data member:

```
      db=new DatabaseHelper(getActivity());
```

When you are done with the database (e.g., your activity is being closed), simply call `close()` on your `SQLiteOpenHelper` to release your connection, as `ConstantsFragment` does (among other things) in `onDestroy()`:

```
@Override
public void onDestroy() {
  super.onDestroy();

  ((CursorAdapter)getListAdapter()).getCursor().close();
  db.close();
}
```

Where to Hold a Helper

For trivial apps, like the one profiled in this chapter, holding a `SQLiteOpenHelper` in a data member of your one-and-only activity is fine.

If, however, you have multiple components — such as multiple activities – all needing to use the database, you are much better served having a *singleton instance* of your `SQLiteOpenHelper`, compared to having each activity have its own instance.

The reason is threading.

You really should do your database I/O on background threads. Opening a database is cheap, but working with it (queries, inserts, etc.) is not. The `SQLiteDatabase` object managed by `SQLiteOpenHelper` is thread-safe... so long as all threads are using the same instance.

For singleton objects that depend upon a `Context`, like `SQLiteOpenHelper`, rather than create the object using a garden-variety `Context` like an `Activity`, you really should create it with an `Application`. There is a singleton instance of a `Context`, in

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the form of the Application subclass, created in your process moments after it is started. You can retrieve this singleton by calling getApplicationContext() on any other Context. The advantage of using Application is memory leaks: if you put a SQLiteOpenHelper in a singleton, and use, say, an Activity to create it, then the Activity might not be able to be garbage-collected, because the SQLiteOpenHelper keeps a strong reference to it. Since Application is itself a singleton (and, hence, is "pre-leaked", so to speak), the risks of a memory leak diminish significantly.

So, instead of:

```
db=new DatabaseHelper(getActivity());
```

in a fragment, with db as a data member, you might have:

```
db=new DatabaseHelper(getActivity().getApplicationContext());
```

with db as a static data member, shared by multiple activities or other components.

Getting Data Out

One popular thing to do with a database is to get data out of it. Android has a few ways you can execute a query on a SQLiteDatabase (from your SQLiteOpenHelper), along with some classes, like CursorAdapter, to help you use the results you get back.

Your Query Options

In most cases, your simplest option for executing a query is to call rawQuery() on the SQLiteDatabase. This takes two parameters:

- A SQL SELECT statement (or anything else that returns a result set), optionally with ? characters in the WHERE clause (or ORDER BY or similar clauses) representing parameters to be bound at runtime
- An optional String array of the parameters to be used to replace the ? characters in the query

If you do not use the ? position parameter syntax in your query, you are welcome to pass null as the second parameter to rawQuery().

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The nice thing about `rawQuery()` is that any valid SQL syntax works, so long as it returns a result set. You are welcome to use joins, sub-selects, and so on without issue.

There are two other query options — `query()` and `SQLiteQueryBuilder`. These both build up a SQL `SELECT` statement from its component parts (e.g., name of the table to query, `WHERE` clause and positional parameters). These are more cumbersome to use, particularly with complex `SELECT` statements. Mostly, they would be used in cases where, for one reason or another, you do not know the precise query at compile time and find it easier to use these facilities to construct the query from parts at runtime. Some developers will do this to avoid duplicating values, by defining constants for things like table names and column names.

For example, `ConstantsFragment` has a private inner class named `BaseTask` which has `doQuery()` method that uses `query()`:

```
abstract private class BaseTask<T> extends AsyncTask<T, Void, Cursor> {
  @Override
  public void onPostExecute(Cursor result) {
    ((CursorAdapter)getListAdapter()).changeCursor(result);
  }

  protected Cursor doQuery() {
    Cursor result=
        db
            .getReadableDatabase()
            .query(DatabaseHelper.TABLE,
                new String[] {"ROWID AS _id",
                              DatabaseHelper.TITLE,
                              DatabaseHelper.VALUE},
                null, null, null, null, DatabaseHelper.TITLE);

    result.getCount();

    return(result);
  }
}
```

Do not concatenate your own `WHERE` clause, though. Let the `?` positional parameters handle that for you, as the work they do to escape your apostrophes, quotation marks, and the like also helps to defend against SQL injection attacks. In this particular case, we do not have a `WHERE` clause.

If that `ROWID AS _id` piece looks a bit odd, we will see why that is in the query a bit later in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What Is a Cursor?

All three of these give you a `Cursor` when you are done. In Android, a `Cursor` represents the entire result set of the query — all the rows and all the columns that the query returned. In this respect, it is reminiscent of a "client-side cursor" from toolkits like ODBC, JDBC, etc.

(if the `Cursor` result set is over 1MB, it actually only holds a "window" on the data, and the story gets really really complicated...)

As such, a `Cursor` can be quite the memory hog. Please `close()` the `Cursor` when you are done with it, to free up the heap space it consumes and make that memory available to the rest of your application.

Using the Cursor Manually

With the `Cursor`, you can:

1. Find out how many rows are in the result set via `getCount()`
2. Iterate over the rows via `moveToFirst()`, `moveToNext()`, and `isAfterLast()`
3. Find out the names of the columns via `getColumnNames()`, convert those into column numbers via `getColumnIndex()`, and get values for the current row for a given column via methods like `getString()`, `getInt()`, etc.

For example, here we iterate over a fictitious `widgets` table's rows:

```
Cursor result=
  db.rawQuery("SELECT _id, name, inventory FROM widgets", null);

while (result.moveToNext()) {
  int id=result.getInt(0);
  String name=result.getString(1);
  int inventory=result.getInt(2);

  // do something useful with these
}

result.close();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Introducing CursorAdapter

Another way to use a `Cursor` is to wrap it in a `CursorAdapter`. Just as `ArrayAdapter` adapts arrays, `CursorAdapter` adapts `Cursor` objects, making their data available to an `AdapterView` like a `ListView`.

The easiest way to set one of these up is to use `SimpleCursorAdapter`, which extends `CursorAdapter` and provides some boilerplate logic for taking values out of columns and putting them into row `View` objects for a `ListView` (or other `AdapterView`). The sample app does just that:

```
    SimpleCursorAdapter adapter=
        new SimpleCursorAdapter(getActivity(), R.layout.row,
            current, new String[] {
            DatabaseHelper.TITLE,
            DatabaseHelper.VALUE },
            new int[] { R.id.title, R.id.value },
            0);

    setListAdapter(adapter);
```

Here, we are telling `SimpleCursorAdapter` to take rows out of a `Cursor` named `constantsCursor`, turning each into an inflated `R.layout.row` `ViewGroup`, in this case, a `RelativeLayout` holding a pair of `TextView` widgets:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

  <TextView
    android:id="@+id/title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:textSize="20sp"
    android:textStyle="bold"/>

  <TextView
    android:id="@+id/value"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentRight="true"
    android:textSize="20sp"
    android:textStyle="bold"/>

</RelativeLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For each row in the `Cursor`, the columns named `title` and `value` (represented by `TITLE` and `VALUE` constants on `DatabaseHelper`) are to be poured into their respective `TextView` widgets (`R.id.title` and `R.id.value`).

Note, though, that if you are going to use `CursorAdapter` or its subclasses (like `SimpleCursorAdapter`), your result set of your query *must* contain an integer column named `_id` that is unique for the result set. This "id" value is then supplied to methods like `onListItemClick()`, to identify what item the user clicked upon in the `AdapterView`. Note that this requirement is on the result set in the `Cursor`, so if you have a suitable column in a table that is not named `_id`, you can rename it in your query (e.g., `SELECT key AS _id, ...`).

However, if you want, you can use the built-in `ROWID`

Quoting the SQLite documentation:

> In SQLite, every row of every table has an 64-bit signed integer ROWID. The ROWID for each row is unique among all rows in the same table. You can access the ROWID of an SQLite table using one the special column names ROWID, _ROWID_, or OID... If a table contains a column of type INTEGER PRIMARY KEY, then that column becomes an alias for the ROWID. You can then access the ROWID using any of four different names, the original three names described above or the name given to the INTEGER PRIMARY KEY column. All these names are aliases for one another and work equally well in any context.

With that in mind, if you want to query SQLite and use the results in a `CursorAdapter`, but you do not have your own `INTEGER PRIMARY KEY` column, you can just include `ROWID` in your query, renaming it to `_id` to satisfy `CursorAdapter`.

That is why we have the `ROWID AS _id` in the `doQuery()` method: to satisfy this `_id` requirement of `CursorAdapter`.

Also note that you cannot `close()` the `Cursor` used by a `CursorAdapter` until you no longer need the `CursorAdapter`. That is why we do not close the `Cursor` until `onDestroy()` of the fragment:

```
  @Override
  public void onDestroy() {
    super.onDestroy();

    ((CursorAdapter)getListAdapter()).getCursor().close();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    db.close();
  }
```

We retrieve the `Cursor` from the `CursorAdapter`, which we get by calling `getListAdapter()` on the fragment.

Getting Data Out, Asynchronously

Ideally, queries are done on a background thread, as they may take some time.

One approach for doing that is to use an `AsyncTask`. In the sample application, `ConstantsFragment` kicks off a `LoadCursorTask` in `onViewCreated()` (shown above). `LoadCursorTask` extends the `BaseTask` class mentioned previously, where the `doQuery()` method resides. `LoadCursorTask` is responsible for doing the query (via the `doQuery()` method shown above) and putting the results in the `ListView` inside the fragment (using the `SimpleCursorAdapter` shown above):

```
abstract private class BaseTask<T> extends AsyncTask<T, Void, Cursor> {
  @Override
  public void onPostExecute(Cursor result) {
    ((CursorAdapter)getListAdapter()).changeCursor(result);
  }

  protected Cursor doQuery() {
    Cursor result=
        db
            .getReadableDatabase()
            .query(DatabaseHelper.TABLE,
                new String[] {"ROWID AS _id",
                              DatabaseHelper.TITLE,
                              DatabaseHelper.VALUE},
                null, null, null, null, DatabaseHelper.TITLE);

    result.getCount();

    return(result);
  }
}

private class LoadCursorTask extends BaseTask<Void> {
  @Override
  protected Cursor doInBackground(Void... params) {
    return(doQuery());
  }
}
```

We execute the actual query in `doInBackground()` and call `getCount()` on the `Cursor`, to force it to actually perform the query — `query()` returns the `Cursor`, but

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the query is not actually executed until we do something that needs the result set. This also holds true for `rawQuery()`, which is why we need to make sure to "touch" the `Cursor` while we are on the background thread.

`onPostExecute()` then uses `changeCursor()` to replace the `Cursor` in the `SimpleCursorAdapter` with the results. Since our `SimpleCursorAdapter` was created with a `null` `Cursor`, `changeCursor()` just slides in the new `Cursor`, telling the `ListView` that the data changed. This causes our `ListView` to be populated.

This way, the UI will not be frozen while the query is being executed, yet we only update the UI from the main application thread.

Also note that the first time we try using the `SQLiteOpenHelper` is in our background thread. `SQLiteOpenHelper` will not try creating our database (e.g., for a new app install) until we call `getReadableDatabase()` or `getWritableDatabase()`. Hence, `onCreate()` (or, later, `onUpgrade()`) of our `SQLiteOpenHelper` will wind up being called on the background thread as well, meaning that the time spent creating (or upgrading) the database also does not freeze the UI.

The Rest of the CRUD

To get data out of a database, it is generally useful to put data into it in the first place. The sample app starts by loading in data when the database is created (in `onCreate()` of `DatabaseHelper`), plus has an action bar item to allow the user to add other constants as needed.

In this section, we will examine in further detail how we manipulate the database, for both the write aspects of CRUD (create-read-update-delete) and for data definition language (DDL) operations (creating tables, creating indexes, etc.).

The Primary Option: execSQL()

For creating your tables and indexes, you will need to call `execSQL()` on your `SQLiteDatabase`, providing the DDL statement you wish to apply against the database. Barring a database error, this method returns nothing.

So, for example, you can call `execSQL()` to create the `constants` table, as shown in the `DatabaseHelper` `onCreate()` method:

```
    db.execSQL("CREATE TABLE constants (title TEXT, value REAL);");
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This will create a table, named `constants`, with two data columns: `title` (text) and `value` (a float, or "real" in SQLite terms).

Most likely, you will create tables and indexes when you first create the database, or possibly when the database needs upgrading to accommodate a new release of your application. If you do not change your table schemas, you might never drop your tables or indexes, but if you do, just use `execSQL()` to invoke `DROP INDEX` and `DROP TABLE` statements as needed.

Alternative Options

For inserts, updates, and deletes of data, you have two choices. You can always use `execSQL()`, just like you did for creating the tables. The `execSQL()` method works for any SQL that does not return results, so it can handle `INSERT`, `UPDATE`, `DELETE`, etc. just fine.

Your alternative is to use the `insert()`, `update()`, and `delete()` methods on the `SQLiteDatabase` object, which eliminate much of the SQL syntax required to do basic operations.

For example, here we `insert()` a new row into our `constants` table, again from `onCreate()` of `DatabaseHelper`:

```
    ContentValues cv=new ContentValues();

    cv.put(TITLE, "Gravity, Death Star I");
    cv.put(VALUE, SensorManager.GRAVITY_DEATH_STAR_I);
    db.insert(TABLE, TITLE, cv);
```

These methods make use of `ContentValues` objects, which implement a `Map`-esque interface, albeit one that has additional methods for working with SQLite types. For example, in addition to `get()` to retrieve a value by its key, you have `getAsInteger()`, `getAsString()`, and so forth.

The `insert()` method takes the name of the table, the name of one column as the "null column hack", and a `ContentValues` with the initial values you want put into this row. The "null column hack" is for the case where the `ContentValues` instance is empty — the column named as the "null column hack" will be explicitly assigned the value `NULL` in the SQL `INSERT` statement generated by `insert()`. This is required due to a quirk in SQLite's support for the SQL `INSERT` statement.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `update()` method takes the name of the table, a `ContentValues` representing the columns and replacement values to use, an optional `WHERE` clause, and an optional list of parameters to fill into the `WHERE` clause, to replace any embedded question marks (`?`). Since `update()` only replaces columns with fixed values, versus ones computed based on other information, you may need to use `execSQL()` to accomplish some ends. The `WHERE` clause and parameter list works akin to the positional SQL parameters you may be used to from other SQL APIs.

The `delete()` method works akin to `update()`, taking the name of the table, the optional `WHERE` clause, and the corresponding parameters to fill into the `WHERE` clause.

Asynchronous CRUD and UI Updates

Just as querying a database should be done on a background thread, so should modifying a database. This is why it is important to make the first time you request a `SQLiteDatabase` from a `SQLiteOpenHelper` be on a background thread, in case `onCreate()` or `onUpgrade()` are needed.

The same thing holds true if you need to update the database during normal operation of your app. For example, the sample application has an “add” action bar item in the upper-right corner of the screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

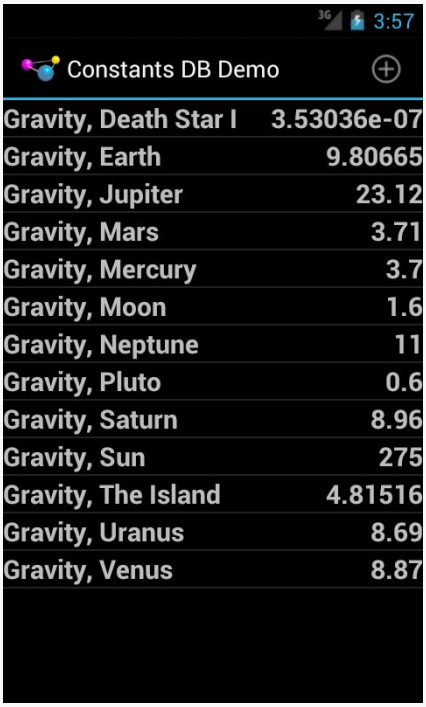


Figure 271: The ConstantsBrowser Sample

Clicking on that brings up a dialog — a technique we will discuss later in this book:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

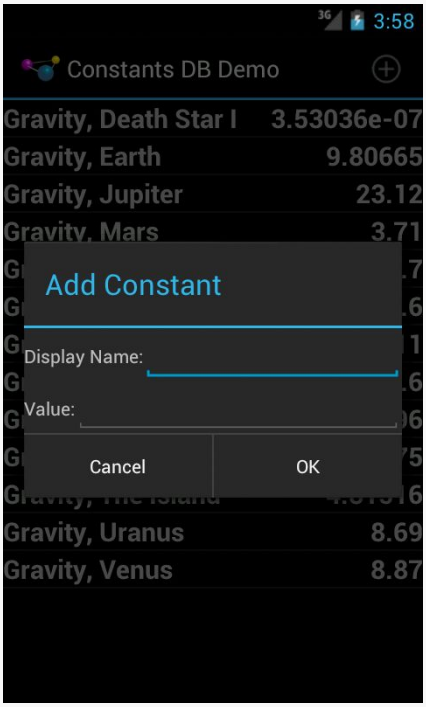


Figure 272: The ConstantsBrowser Sample, Add Constant Dialog

If the user fills in a constant and clicks the "OK" button, we need to insert a new record in the database. That is handled via an `InsertTask`:

```
  private class InsertTask extends BaseTask<ContentValues> {
    @Override
    protected Cursor doInBackground(ContentValues... values) {
      db.getWritableDatabase().insert(DatabaseHelper.TABLE,
                                      DatabaseHelper.TITLE, values[0]);

      return(doQuery());
    }
  }
```

The `InsertTask` is supplied a `ContentValues` object with our `title` and `value`, just as we used in `onCreate()` of `DatabaseHelper`. In `doInBackground()`, we get a writable database from `DatabaseHelper` and perform the `insert()` call, so the database I/O does not tie up the main application thread.

However, in `doInBackground()`, we *also* call `doQuery()` again. This retrieves a fresh `Cursor` with the new roster of constants... including the one we just inserted. As with `LoadCursorTask`, we execute `doQuery()` in `doInBackground()` to keep the database I/O off the main application thread. This triggers the same `onPostExecute()` as

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

before, inherited from `BaseTask`, which uses `changeCursor()` to replace any existing results with the new results.

Setting Transaction Bounds

By default, each SQL statement executes in its own transaction — this is fairly typical behavior for a SQL database, and SQLite is no exception.

There are two reasons why you might want to have your own transaction bounds, larger than a single statement:

1. The classic "we have a series of operations that need to succeed or fail as a whole" rationale, for maintaining data integrity
2. Performance, as each database transaction involves disk I/O, and one large transaction will be much faster than lots of little transactions

The basic recipe for your own transactions is:

```
try {
  db.beginTransaction();

  // several SQL statements in here

  db.setTransactionSuccessful();
}
finally {
  db.endTransaction();
}
```

`beginTransaction()` marks the fact that you want a transaction. `setTransactionSuccessful()` indicates that you want the transaction to commit. However, the actual `COMMIT` or `ROLLBACK` does not occur until `endTransaction()`. In the normal case, `setTransactionSuccessful()` does get called, and `endTransaction()` performs a `COMMIT`. If, however, one of your SQL statements fails (e.g., violates a foreign key constraint), the `setTransactionSuccessful()` call is skipped, so `endTransaction()` will do a `ROLLBACK`.

You might wonder why we did not bother with a transaction in `onCreate()` method of `DatabaseHelper`, given the latter reason. That is because `onCreate()` is called within a transaction set up by `SQLiteOpenHelper` itself, so you do not need your own.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Hey, What About Hibernate?

Those of you with significant Java backgrounds outside of Android are probably pounding your head against your desk right about now. Outside of a few conveniences like `SQLiteOpenHelper` and `CursorAdapter`, Android's approach to database I/O feels a bit like classic JDBC. Java developers, having experienced the pain of raw JDBC, created various wrappers around it, the most prominent of which is an ORM (object-relational mapper) called Hibernate.

Alas, Hibernate is designed for servers, not mobile devices. It is a little bit heavyweight, and it is designed for use with JDBC, not Android's SQLite classes.

Android did not include any sort of ORM in the beginning for two main reasons:

1. To keep the firmware size as small as possible, as smaller firmware can lead to less-expensive devices
2. To eliminate the ORM overhead (e.g., reflection), which would have been too much for early Android versions on early Android devices

The Android ecosystem has come up with alternatives, such as ORMLite and greenDAO. So, if you are used to using an ORM, you may want to investigate these sorts of solutions — they just are not built into Android itself.

Visit the Trails!

If you are interested in exposing your database contents to a third-party application, you may wish to read up on `ContentProvider`.

The trails also have chapters on encrypted databases using SQLCipher and shipping pre-packaged databases with your app.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #14 - Saving Notes

It would be nice if the user could add some personal notes to the chapter that she is reading, whether that serves as commentary, points to be researched, complaints about the author's hair (or lack thereof), or whatever.

So, in this chapter, we will add a new fragment and new activity to allow the user to add notes per chapter, via a large `EditText` widget. Those notes will be stored in a SQLite database.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Adding a DatabaseHelper

The first step for working with SQLite is to add an implementation of `SQLiteOpenHelper`, which we will do here, named `DatabaseHelper`.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `DatabaseHelper` as the name and click OK to create the empty class.

Then, replace the contents of that class with the following:

```
package com.commonsware.empublite;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class DatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="empublite.db";
  private static final int SCHEMA_VERSION=1;
  private static DatabaseHelper singleton=null;

  synchronized static DatabaseHelper getInstance(Context ctxt) {
    if (singleton == null) {
      singleton=new DatabaseHelper(ctxt.getApplicationContext());
    }

    return(singleton);
  }

  private DatabaseHelper(Context ctxt) {
    super(ctxt, DATABASE_NAME, null, SCHEMA_VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE notes (position INTEGER PRIMARY KEY, prose TEXT);");
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion,
                        int newVersion) {
    throw new RuntimeException("This should not be called");
  }
}
```

Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. Fill in `DatabaseHelper` in the “Name” field. Click the “Browse...” button next to the “Superclass” field and find

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`SQLiteOpenHelper` to set as the superclass. Then, click "Finish" on the new-class dialog to create the `DatabaseHelper` class.

Then, with `DatabaseHelper` open in the editor, paste in the class definition shown in the Android Studio instructions above.

Step #2: Examining DatabaseHelper

Our initial version of `DatabaseHelper` has a few things:

- It has the constructor, supplying to the superclass the name of the database file (`DATABASE_NAME`) and the revision number of our schema (`SCHEMA_VERSION`). Note that the constructor is private, as we are using the singleton pattern, so only `DatabaseHelper` should be able to create `DatabaseHelper` instances.
- It has the `onCreate()` method, invoked the first time we run the app on a device or emulator, to let us populate the database. Here, we use `execSQL()` to define a `notes` with a `position` column (indicating our chapter) and a `prose` column (what the user types in as the note).
- It has the `onUpgrade()` method, needed because `SQLiteOpenHelper` is abstract, so our app will not compile without an implementation. Until we revise our schema, though, this method should never be called, so we raise a `RuntimeException` in the off chance that it *is* called unexpectedly.
- It has a static `DatabaseHelper` singleton instance and a `getInstance()` method to lazy-initialize it.

As noted in the chapter on databases, it is important to ensure that all threads are accessing the same `SQLiteDatabase` object, for thread safety. That usually means you hold onto a single `SQLiteOpenHelper` object. And, in our case, we might want to get at this database from more than one activity. Hence, we go with the singleton approach, so everyone works with the same `DatabaseHelper` instance.

Step #3: Creating a NoteFragment

Having a database is nice and all, but we need to work on the UI to allow users to enter notes. To do that, we will start with a `NoteFragment`.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `NoteFragment` as the name and click OK to create the empty class.

Next, replace the contents of that class with the following:

```
package com.commonsware.empublite;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;

public class NoteFragment extends Fragment {
  private static final String KEY_POSITION="position";
  private EditText editor=null;

  static NoteFragment newInstance(int position) {
    NoteFragment frag=new NoteFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_POSITION, position);
    frag.setArguments(args);

    return(frag);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.editor, container, false);

    editor=(EditText)result.findViewById(R.id.editor);

    return(result);
  }

  private int getPosition() {
    return(getArguments().getInt(KEY_POSITION, -1));
  }
}
```

You will have a couple of warnings, for things in this code that we are setting up (`editor` and `position`) that are not yet used, but we will take care of that in later steps of the tutorial.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Then, right-click over the `res/` directory and choose New > XML > Layout XML File from the context menu. This brings up the New Layout XML File dialog:

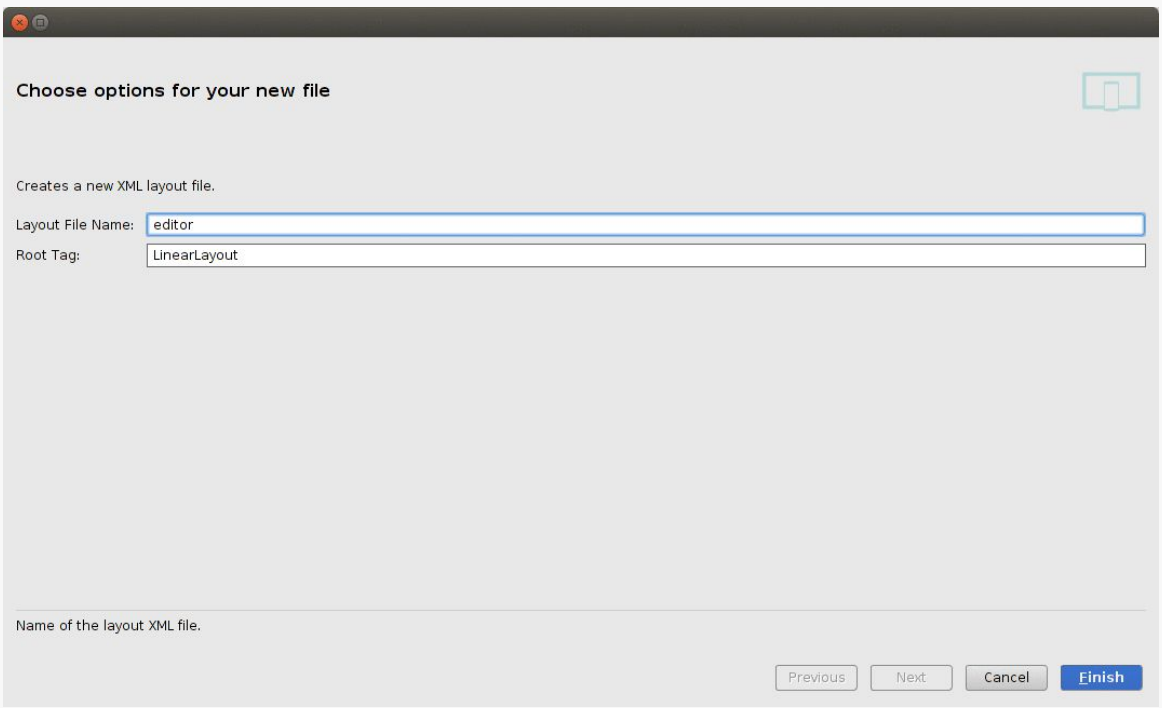


Figure 273: Android Studio New Layout XML File Dialog

Fill in `editor` as the "Layout File Name", leave the rest of the dialog alone, and click the "Finish" button.

In the Design tab of the graphical layout editor for `res/layout/editor.xml`, drag a "Multiline Text" widget from the palette into the preview area. In the properties pane, change the `android:layout_width` and `android:layout_height` each to be `match_parent`, change the ID to `editor`, and change the gravity (*not* the layout gravity) to be both `top` and `left`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

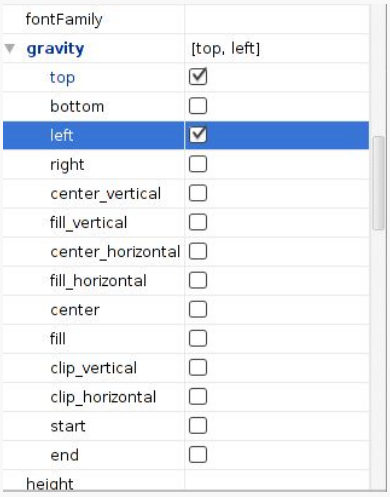


Figure 274: Android Studio Layout Editor Properties Pane, Showing Gravity Options

Next, in the properties pane, click on the "hint" entry, then click the "..." button to the right of it. This will open up a string resource picker dialog:

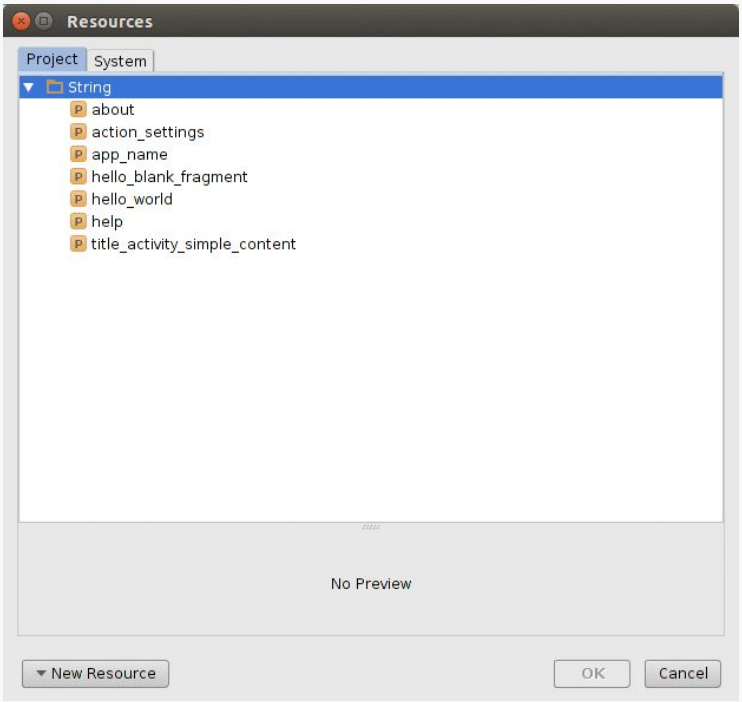


Figure 275: Android Studio String Resource Picker Dialog

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Towards the bottom, click the "New Resource" drop-down and choose "New String Value..." from it, to bring up the string resource editor dialog:

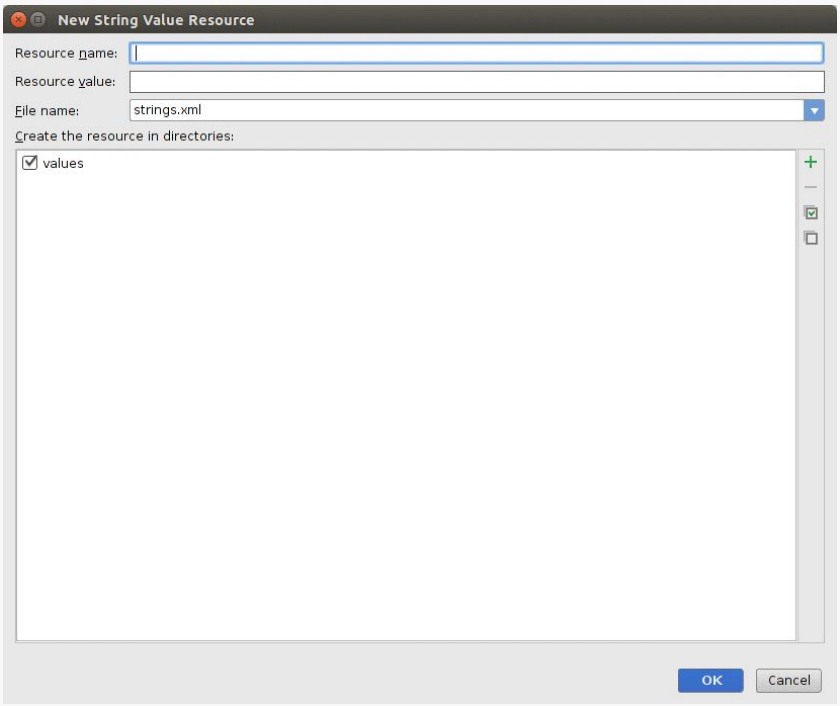


Figure 276: Android Studio New String Resource Dialog

Fill in a resource name of `hint` and a value of `Enter notes here`. Leave the rest of the dialog alone, and click OK.

Eclipse

Right click over the `res/layout/` directory in your project, and choose New > Other from the context menu. Choose "Android XML Layout File" from the first page of the new-file wizard, then click "Next". In the second page of the wizard, give the file a name of `editor.xml` and choose `LinearLayout` as being the root element. Then, click "Finish" to generate the file.

Next, in the Graphical Layout sub-tab of the Eclipse layout editor, click on the "Text Fields" section of the tool palette, and drag a "Multiline Text" widget into the layout. Give it an ID of `@+id/editor`. Change the "Layout height" and "Layout width" to be `match_parent`. Add or change the Gravity to be `top|left`. Finally, add or change the Hint to be a new string resource (named `hint`) with a value of `Enter notes here`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Then, right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. Fill in `NoteFragment` in the "Name" field. Click the "Browse..." button next to the "Superclass" field and find `Fragment - (android.app)` to set as the superclass. Then, click "Finish" on the new-class dialog to create the `NoteFragment` class.

Then, with `NoteFragment` open in the editor, paste in the class definition seen in the Android Studio instructions above. As is noted there, you will have a couple of warnings, for things in this code that we are setting up (`editor` and `position`) that are not yet used, but we will take care of that in later steps of the tutorial.

Step #4: Examining NoteFragment

Our `NoteFragment` is fairly straightforward and is reminiscent of the `SimpleContentFragment` we created in Tutorial #11.

`NoteFragment` has a `newInstance()` static factory method. This method creates an instance of `NoteFragment`, takes a passed-in `int` (identifying the chapter for which we are creating a note), puts it in a `Bundle` identified as `KEY_POSITION`, hands the `Bundle` to the fragment as its arguments, and returns the newly-created `NoteFragment`.

In `onCreateView()`, we inflate the `R.layout.editor` resource that we defined and get our hands on our `EditText` widget for later use.

Step #5: Creating the NoteActivity

Having a fragment without displaying it is fairly pointless, so we need something to load a `NoteFragment`. Particularly for phones, the simplest answer is to create a `NoteActivity` for that, paralleling the relationship between `SimpleContentFragment` and `SimpleContentActivity`.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `NoteActivity` as the name and click OK to create the empty class.

In the `NoteActivity` class that is created, replace the current implementation with the following:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.Fragment;
import android.os.Bundle;

public class NoteActivity extends Activity {
  public static final String EXTRA_POSITION="position";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      int position=getIntent().getIntExtra(EXTRA_POSITION, -1);

      if (position >= 0) {
        Fragment f=NoteFragment.newInstance(position);

        getFragmentManager().beginTransaction()
                            .add(android.R.id.content, f).commit();
      }
    }
  }
}
```

Then, open up `AndroidManifest.xml` and add the following `<activity>` element inside the `<application>` element:

```
<activity android:name="NoteActivity"></activity>
```

Eclipse

Right-click over the EmPubLite project in the Package Explorer, then choose New > Other from the context menu. In the list of possible things to add to your project, choose “Android Activity” and click Next. In the list of templates, choose “Empty Activity” and click Next.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In the last page of the new-activity wizard, fill in:

- `NoteActivity` in the "Activity Name" field
- `pleasedeleteme` for the "Layout Name"
- `pleasedeletemetoo` for the "Title"

Leave the rest of the wizard at its defaults, then click on Finish.

In the `NoteActivity` class that is created, replace the current implementation with the source shown in the Android Studio instructions above.

You can also delete the `/res/layout/pleasedeleteme.xml` file, by right-clicking over it and choosing "Delete..." from the context menu.

Finally, open up `res/values/strings.xml` and delete the `title_activity_simple_content` that the wizard added, which we are no longer using. At this point, you can save your changes to this file (e.g., `<Ctrl>-<S>`).

Step #6: Examining NoteActivity

As you can see, this is a fairly trivial activity. In `onCreate()`, if we are being created anew, we execute a `FragmentTransaction` to add a `NoteFragment` to our activity, pouring it into the full screen (`android.R.id.content`). Here, `android.R.id.content` identifies the container into which the results of `setContentView()` would go — it is a container supplied by Activity itself and serves as the top-most container for our content.

However, we expect that we will be passed an `Intent` extra with the position (`EXTRA_POSITION`), which we pass along to the `NoteFragment` factory method.

Step #7: Add Notes to the Action Bar

Of course, none of this is useful if we do not give the user a way to get to the `NoteActivity`. Specifically, we can add a `notes` entry to our `res/menu/options.xml` resource, to have a new toolbar button appear on our main activity's action bar.

First, copy the `01_core_edit` icons from the `Action Bar Icons/holo_dark/` directory from the "Action Bar Icon Pack" into your project. In the `01_core_edit` directory, you will see versions of this icon in various densities. Copy those icon files into their corresponding directories in your project. You can do this by dragging the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

file from your development desktop OS into the project explorer window and dropping it into the proper directory.

Then, modify `res/menu/options.xml` to look like:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/notes"
        android:icon="@drawable/ic_action_edit"
        android:showAsAction="ifRoom|withText"
        android:title="@string/notes">
    </item>
    <item
        android:id="@+id/settings"
        android:icon="@drawable/ic_action_settings"
        android:showAsAction="never"
        android:title="@string/settings">
    </item>
    <item
        android:id="@+id/help"
        android:icon="@drawable/ic_action_help"
        android:title="@string/help">
    </item>
    <item
        android:id="@+id/about"
        android:icon="@drawable/ic_action_about"
        android:title="@string/about">
    </item>

</menu>
```

Either IDE can handle this via editing the XML. Eclipse users also can add this via the structured editor for `res/menu/options.xml`, following the instructions used for other action bar items.

Note that this menu definition requires a new string resource, named `notes`, with a value like `Notes`.

Then, in `EmPubLiteActivity`, add the following `case` to the `switch` statement in `onOptionsItemSelected()`:

```
      case R.id.notes:
        i=new Intent(this, NoteActivity.class);
        i.putExtra(NoteActivity.EXTRA_POSITION, pager.getCurrentItem());
        startActivity(i);

        return(true);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that depending on where you place this, you will need to remove one existing declaration of `Intent i` from one of the `case` blocks, whichever comes second.

Here, we get the currently-viewed position from the `ViewPager` and pass that as the `EXTRA_POSITION` extra to `NoteActivity`.

Step #8: Defining a NoteLoadedEvent

We will want to load notes from the database on a background thread. Hence, we can apply the same basic approach as we used with `ModelFragment`, posting an event on the greenrobot EventBus when the load is completed, to deliver the results to the `NoteFragment`. This step will create a `NoteLoadedEvent` to handle this case.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `NoteLoadedEvent` as the name and click OK to create the empty class.

Then, replace the contents of that class with the following:

```
package com.commonsware.empublite;

class NoteLoadedEvent {
  int position;
  String prose;

  NoteLoadedEvent(int position, String prose) {
    this.position=position;
    this.prose=prose;
  }

  int getPosition() {
    return(position);
  }

  String getProse() {
    return(prose);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. Fill in `NoteLoadedEvent` in the "Name" field. Leave the "Superclass" field alone, as `NoteLoadedEvent` has no explicit superclass. Then, click "Finish" on the new-class dialog to create the `NoteLoadedEvent` class.

Then, with `NoteLoadedEvent` open in the editor, paste in the class definition shown in the Android Studio instructions above.

Step #9: Loading a Note from the Database

Next, we need to add code somewhere that will actually query the database (on a background thread) to load the note for a given `ViewPager` position. One common pattern is to put this sort of database-access logic on your `SQLiteOpenHelper` subclass, so all of your database-specific code resides in one place. That is the approach we will take here, adding a `loadNote()` method that will fork a thread, query the database, and post a `NoteLoadedEvent` as a result.

Edit your `DatabaseHelper` to add its own `LoadThread` inner class, reminiscent of the one from `ModelFragment`:

```
  private class LoadThread extends Thread {
    private int position=-1;

    LoadThread(int position) {
      super();

      this.position=position;
      Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
    }

    @Override
    public void run() {
      String[] args= { String.valueOf(position) };
      Cursor c=
          getReadableDatabase().rawQuery("SELECT prose FROM notes WHERE position
= ?",
                                         args);
      if (c.getCount() > 0) {
        c.moveToFirst();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      EventBus.getDefault().post(new NoteLoadedEvent(position,
                                                     c.getString(0)));
    }

    c.close();
  }
}
```

Here, we use `rawQuery()` to retrieve the note based upon a supplied position. If there is no such note, our `Cursor` will have no rows, and we are done. If, however, we did get results back on the query, we then post a `NoteLoadedEvent` with the `position` and the `prose` (the text from the database).

You will need to add an import manually to `android.os.Process`, to be able to resolve the `setThreadPriority()` method and its parameter.

Also, add a `loadNote()` method to `DatabaseHelper` that forks this `LoadThread`:

```
void loadNote(int position) {
  new LoadThread(position).start();
}
```

Step #10: Loading the Note Into the Fragment

Now that we can query the database and get back a note (if any), we can tie that into the `NoteFragment` to load the note for the fragment's `position` when the fragment is opened. We will not only need to call `loadNote()` on the `DatabaseHelper`, but also be able to respond to the `NoteLoadedEvent` when it arrives.

Add the following `onResume()` method to `NoteFragment`:

```
@Override
public void onResume() {
  super.onResume();

  EventBus.getDefault().register(this);

  if (TextUtils.isEmpty(editor.getText())) {
    DatabaseHelper db=DatabaseHelper.getInstance(getActivity());

    db.loadNote(getPosition());
  }
}
```

Here, we register for the `EventBus`. Then, if we do not have any text in the `EditText` widget, we call `loadNote()` on our singleton instance of the `DatabaseHelper`, passing

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

in the `position` that our fragment is managing. The reason for checking to see if the `EditText` is empty is to handle configuration changes. This fragment is not a retained fragment, and so it will be destroyed and re-created. The default `onSaveInstanceState()` logic of `EditText` will retain our note, though, so we do not want to re-load it from the database. This approach is not optimal, in that we will wind up calling `loadNote()` in cases where we could know that there is no note. That optimization is complex enough to not make it worthwhile for a set of book tutorials, though it is something you might wish to explore in a commercial-grade application.

Next, add the corresponding `onPause()` to `NoteFragment`, to unregister from the `EventBus`:

```
  @Override
  public void onPause() {
    EventBus.getDefault().unregister(this);

    super.onPause();
  }
```

Finally, add an `onEventMainThread(NoteLoadedEvent)` method to `NoteFragment`, so we receive the `NoteLoadedEvent` on the main application thread:

```
  public void onEventMainThread(NoteLoadedEvent event) {
    if (event.getPosition() == getPosition()) {
      editor.setText(event.getProse());
    }
  }
```

Here, we confirm that the event is for our fragment's position, as it is conceivable that this event is for some other note, though that is rather unlikely given how the user would view notes. That being said, if the note is for our position, we populate the `EditText` with the note prose.

Step #11: Updating the Database

Of course, loading notes from a database is all fine and well... except that we do not have any notes *in* the database. We really should fix that.

Add an `UpdateThread` inner class to `DatabaseHelper`:

```
  private class UpdateThread extends Thread {
    private int position=-1;
    private String prose=null;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    UpdateThread(int position, String prose) {
      super();

      this.position=position;
      this.prose=prose;
      Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
    }

    @Override
    public void run() {
      String[] args= { String.valueOf(position), prose };

      getWritableDatabase().execSQL("INSERT OR REPLACE INTO notes (position,
prose) VALUES (?, ?)",
                                    args);
    }
  }
```

Here, we use `execSQL()` to execute an `INSERT OR REPLACE` SQL statement. As the name suggests, this will insert a new row if there is no match on our primary key (`position`). Otherwise, it will update the other columns if there is a match.

Note that we do not post an event here. We could, if there was something in the app that needed to know when a note was updated.

Also, add an `updateNote()` method to `DatabaseHelper` that forks this `UpdateThread`:

```
  void updateNote(int position, String prose) {
    new UpdateThread(position, prose).start();
  }
```

Step #12: Saving the Note

Somewhere, we need to call `updateNote()`. A classic "desktop" approach would be to have a "save" action bar item in the `NoteFragment`, which the user would need to click upon to save the note. However, this does not deal with the interrupt-driven nature of phones all that well. For example, the user might start typing in a note, then wind up taking a phone call. If our process is terminated, depending upon how the user tries getting back into our app, we might not have the note from our saved instance state.

A better approach, in many cases, is to save data in `onPause()` or `onStop()`, when the activity moves into the background. If there is a chance that the user might not want the partially-entered information, you could save it in a "side" area, such as a

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

temporary file, and deal with it when the user returns to your app. Or, you could just update the real data store... which is what we will do here.

Edit the `onPause()` method in `NoteFragment` to look like the following:

```
  @Override
  public void onPause() {
    DatabaseHelper.getInstance(getActivity())
                  .updateNote(getPosition(),
                              editor.getText().toString());

    EventBus.getDefault().unregister(this);

    super.onPause();
  }
```

Here, we update the note. This is a bit inefficient, as we update the database even if the user did not change the text of the note, or even if the note is empty. That represents another optimization that a production-grade app might wish to pursue but is skipped here in the interests of simplicity.

If you build and run the app on a device or emulator, you will see the new "notes" toolbar button in the action bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

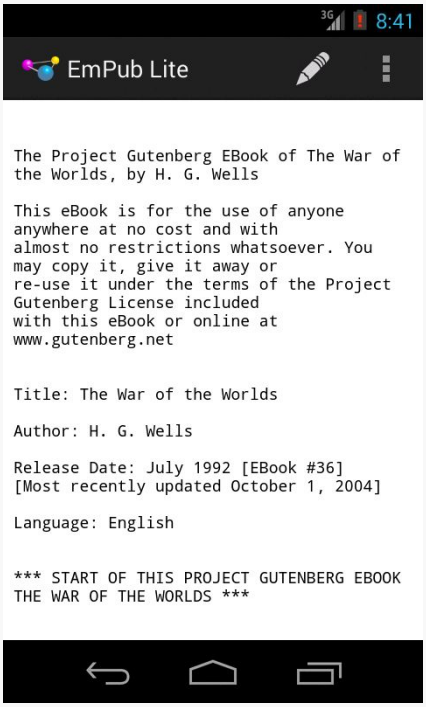


Figure 277: The New Action Bar Item

Tapping that will bring up the notes for whatever `ViewPager` position that you are on. Entering in some notes and pressing BACK to exit the activity will save those notes, which you will see again if you tap the action bar toolbar button again. If you change the notes, pressing BACK will save the changed notes in the database, to be viewed again later when you go back into the notes for that `ViewPager` position.

Step #13: Adding a Delete Action Bar Item

The only problem with this solution is that the notes never leave. While the user could manually delete everything in the `EditText`, it would be nice to make that perhaps a bit simpler. In this step, we will add an action bar item that will clear the `EditText` for the user.

First, copy the `01_core_discard` icons from the `Action Bar Icons/holo_dark/` directory from the "Action Bar Icon Pack" into your project. In the `01_core_discard` directory, you will see versions of this icon in various densities. Copy those icon files into their corresponding directories in your project. You can do this by dragging the file from your development desktop OS into the project explorer window and dropping it into the proper directory.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Create a new resource, `res/menu/notes.xml`, to configure the action bar for the activity hosting our `NoteFragment`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/delete"
        android:icon="@drawable/ic_action_discard"
        android:showAsAction="ifRoom|withText"
        android:title="@string/delete">
    </item>
</menu>
```

This simply defines a single action bar item, with an ID of `delete`.

To do this, Android Studio users can right-click over the `res/menu/` directory and choose New > "Menu resource file" from the context menu. Fill in `notes.xml` in the "New Menu Resource File" dialog and click OK. Paste in the XML shown above into that file.

Eclipse users can right-click over the `res/menu/` directory and choose New > File from the context menu, filling in `notes.xml` as the file name. Then, switch over to the "notes.xml" sub-tab and paste in the XML shown above into that file.

If you prefer, you can view this file's contents in your Web browser via this GitHub link. Note that you will also need to add a new string resource, named `delete`, with a value of `Delete`.

To let Android know that our `NoteFragment` wishes to participate in the action bar, we need to call `setHasOptionsMenu(true)` at some point. Add an `onCreate()` method to `NoteFragment` to handle this when the fragment is created:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setHasOptionsMenu(true);
  }
```

That will trigger a call to `onCreateOptionsMenu()`, which we will need to add to `NoteFragment`:

```
  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.notes, menu);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onCreateOptionsMenu(menu, inflater);
  }
```

This just inflates our new resource for use in the options menu.

If the user taps on that toolbar button, `onOptionsItemSelected()` will be called, so we will need to add that as well to `NoteFragment`:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.delete) {
      editor.setText(null);

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

Here, if the user tapped on our `delete` action bar item, we clear the `EditText` widget.

Step #14: Closing the NoteFragment When Deleted

However, tapping on that action bar item keeps the `NoteFragment` on the screen. It might be nice to automatically return to the book instead. However, the `NoteFragment` itself does not know how to do that, as something else (in this case, `NoteActivity`) put the `NoteFragment` on the screen. Hence, we need to pass the request to close the `NoteFragment` along to the proper party.

We could use another event object and our `EventBus`. In this case, we will demonstrate another approach: using the contract pattern to alert the hosting activity that the notes should be closed.

Define an *inner interface* in the `NoteFragment`, named `Contract`, as follows:

```
  public interface Contract {
    void closeNotes();
  }
```

You might put those lines immediately after the `public class NoteFragment...` line, before the declaration of any of the data members or methods, for example.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Then, add a private `getContract()` method, that casts the hosting `Activity` to the `Contract` interface:

```
  private Contract getContract() {
    return((Contract)getActivity());
  }
```

What we are doing here is enforcing that the activity that hosts our `NoteFragment` must implement the `NoteFragment.Contract` interface.

Then, add a call to `closeNotes()` on the `Contract` to our logic in `onOptionsItemSelected()`:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.delete) {
      editor.setText(null);
      getContract().closeNotes();

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

Now, when the user clicks on the `delete` action bar item, we clear the `EditText` and ask the hosting activity to get rid of us. Along the way, our `onPause()` will be called, causing us to clear the content of the `prose` column in our database row as well.

At this point, `NoteFragment` should resemble:

```
package com.commonsware.empublite;

import android.app.Fragment;
import android.os.Bundle;
import android.text.TextUtils;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;
import de.greenrobot.event.EventBus;

public class NoteFragment extends Fragment {
  public interface Contract {
    void closeNotes();
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
private static final String KEY_POSITION="position";
private EditText editor=null;

static NoteFragment newInstance(int position) {
  NoteFragment frag=new NoteFragment();
  Bundle args=new Bundle();

  args.putInt(KEY_POSITION, position);
  frag.setArguments(args);

  return(frag);
}

@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setHasOptionsMenu(true);
}

@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {
  View result=inflater.inflate(R.layout.editor, container, false);

  editor=(EditText)result.findViewById(R.id.editor);

  return(result);
}

@Override
public void onResume() {
  super.onResume();

  EventBus.getDefault().register(this);

  if (TextUtils.isEmpty(editor.getText())) {
    DatabaseHelper db=DatabaseHelper.getInstance(getActivity());

    db.loadNote(getPosition());
  }
}

@Override
public void onPause() {
  DatabaseHelper.getInstance(getActivity())
                .updateNote(getPosition(),
                            editor.getText().toString());

  EventBus.getDefault().unregister(this);

  super.onPause();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }

  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.notes, menu);

    super.onCreateOptionsMenu(menu, inflater);
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.delete) {
      editor.setText(null);
      getContract().closeNotes();

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }

  public void onEventMainThread(NoteLoadedEvent event) {
    if (event.getPosition() == getPosition()) {
      editor.setText(event.getProse());
    }
  }

  private int getPosition() {
    return(getArguments().getInt(KEY_POSITION, -1));
  }

  private Contract getContract() {
    return((Contract)getActivity());
  }
}
```

`NoteActivity` now must implement `NoteFragment.Contract` and implement `closeNotes()`. Modify `NoteActivity` to look like:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.Fragment;
import android.os.Bundle;

public class NoteActivity extends Activity implements
    NoteFragment.Contract {
  public static final String EXTRA_POSITION="position";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      int position=getIntent().getIntExtra(EXTRA_POSITION, -1);

      if (position >= 0) {
        Fragment f=NoteFragment.newInstance(position);

        getFragmentManager().beginTransaction()
                            .add(android.R.id.content, f).commit();
      }
    }
  }

  public void closeNotes() {
    finish();
  }
}
```

This adds the `implements` keyword and the `closeNotes()` implementation, which just finishes the `NoteActivity`, returning control to the `EmPubLiteActivity`.

If you run this in a device or emulator, and you go into the notes, you will see our delete toolbar button:

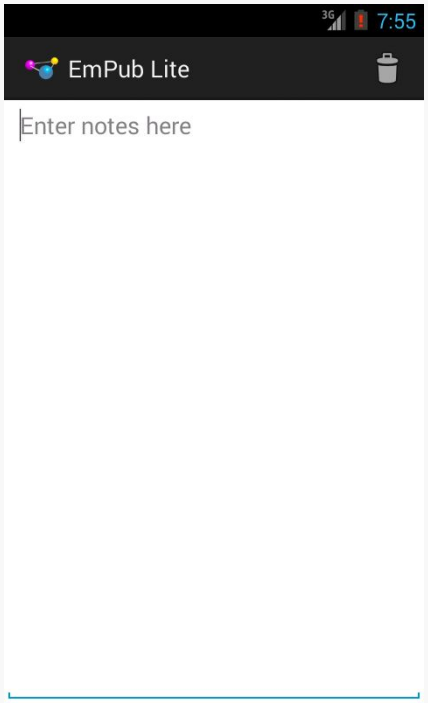


Figure 278: The New Action Bar Item

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tapping that toolbar button will clear the note and close the activity, returning you to the book.

In Our Next Episode…

… we will allow the user to share a chapter's notes with somebody else.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Internet Access

The expectation is that most, if not all, Android devices will have built-in Internet access. That could be WiFi, cellular data services (EDGE, 3G, etc.), or possibly something else entirely. Regardless, most people — or at least those with a data plan or WiFi access — will be able to get to the Internet from their Android phone.

Not surprisingly, the Android platform gives developers a wide range of ways to make use of this Internet access. Some offer high-level access, such as the integrated WebKit browser component (`WebView`) we saw in an earlier chapter. If you want, you can drop all the way down to using raw sockets. Or, in between, you can leverage APIs — both on-device and from 3rd-party JARs — that give you access to specific protocols: HTTP, XMPP, SMTP, and so on.

The emphasis of this book is on the higher-level forms of access: the WebKit component and Internet-access APIs, as busy coders should be trying to reuse existing components versus rolling one's own on-the-wire protocol wherever possible.

DIY HTTP

In many cases, your only viable option for accessing some Web service or other HTTP-based resource is to do the request yourself. The Google-endorsed API for doing this nowadays in Android is to use the classic `java.net` classes for HTTP operation, centered around `HttpUrlConnection`. There is quite a bit of material on this already published, as these classes have been in Java for a long time. The focus here is in showing how this works in an Android context.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note, however, that you may find it easier to use some HTTP client libraries that handle various aspects of the Internet access for you, as will be described later in this chapter.

A Sample Usage of HttpUrlConnection

This chapter walks through several implementations of a Stack Overflow client application. The app has a single activity, with a single `ListFragment`. The app will load the latest block of Stack Overflow questions tagged with `android`, using the Stack Exchange public API. Those questions will be shown in the list, and tapping on a question will bring up the Web page for that question in the user's default Web browser.

All implementations of the app have the same core UI logic. What differs is in how each handles the Internet access. In this section, we will take a look at the `Internet/HURL` sample project, which uses `HttpUrlConnection` to retrieve the questions from the Stack Exchange Web service API.

Asking Permission

To do anything with the Internet (or a local network) from your app, you need to hold the `INTERNET` permission. This includes cases where you use things like `WebView` — if your *process* needs network access, you need the `INTERNET` permission.

Hence, the manifest for our sample project contains the requisite `<uses-permission>` declaration:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Creating Your Data Model

The Stack Exchange Web service API returns JSON in response to various queries. Hence, we need to create Java classes that mirror that JSON structure. In particular, many of the examples will be using Google's Gson to populate those data models automatically based upon its parsing of the JSON that we receive from the Web service.

In our case, we are going to use a specific endpoint of the Stack Exchange API, referred to as `/questions` after the distinguishing portion of the path. The

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

documentation for this endpoint can be found in the Stack Exchange API documentation.

We will examine the URL for the endpoint a bit later in this section.

The results we get for issuing a `GET` request for the URL is a JSON structure (here showing a single question, to keep the listing short):

```
{
  "items": [
    {
      "question_id": 17196927,
      "creation_date": 1371660594,
      "last_activity_date": 1371660594,
      "score": 0,
      "answer_count": 0,
      "title": "ksoap2 failing when in 3G",
      "tags": [
        "android",
        "ksoap2",
        "3g"
      ],
      "view_count": 2,
      "owner": {
        "user_id": 773259,
        "display_name": "SparK",
        "reputation": 513,
        "user_type": "registered",
        "profile_image": "http://www.gravatar.com/avatar/
511b37f7c313984e624dd76e8cb9faa6?d=identicon&r=PG",
        "link": "http://stackoverflow.com/users/773259/spark"
      },
      "link": "http://stackoverflow.com/questions/17196927/
ksoap2-failing-when-in-3g",
      "is_answered": false
    }
  ],
  "quota_remaining": 9991,
  "quota_max": 10000,
  "has_more": true
}
```

NOTE: Some of the longer URLs will word-wrap in the book, but they are on a single line in the actual JSON. Honest.

We get back a JSON object, where our questions are found under the name of `items`. `items` is a JSON array of JSON objects, where each JSON object represents a single question, with fields like `title` and `link`. The question JSON object has an embedded `owner` JSON object with additional information.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We do not necessarily need all of this information. In fact, for this first version of the sample, all we really need are the `title` and `link` of each entry in the `items` array.

The key is that, by default, the data members in our Java data model must *exactly* match the JSON keys for the JSON objects.

So, we have an `Item` class, representing the information from an individual entry in the `items` array:

```
package com.commonsware.android.hurl;

public class Item {
  String title;
  String link;

  @Override
  public String toString() {
    return(title);
  }
}
```

The `toString()` implementation is there so we can toss our `Item` array into an `ArrayAdapter` without needing to write a `getView()` method to pull the `title` out manually.

However, our Web service does not return the `items` array directly. `items` is the key in a JSON object that is the actual JSON returned by Stack Exchange. So, we need another Java class that contains the data members we need from that outer JSON object, here named `SOQuestions` (for lack of a better idea for a name...):

```
package com.commonsware.android.hurl;

import java.util.List;

public class SOQuestions {
  List<Item> items;
}
```

Having an `items` data member that is a `List` of `Item` tells GSON that we are expecting the JSON object to be used for `SOQuestions` to have a JSON array, named `items`, where each element in that array should get mapped to `Item` objects.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A Thread for Loading

We need to do the network I/O on a background thread, so we do not tie up the main application thread. To that end, the sample app has a `LoadThread` that loads our questions:

```
package com.commonsware.android.hurl;

import android.util.Log;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import com.google.gson.Gson;
import de.greenrobot.event.EventBus;

class LoadThread extends Thread {
  static final String SO_URL=
      "https://api.stackexchange.com/2.1/questions?"
          + "order=desc&sort=creation&site=stackoverflow&tagged=android";

  @Override
  public void run() {
    try {
      HttpURLConnection c=
          (HttpURLConnection)new URL(SO_URL).openConnection();

      try {
        InputStream in=c.getInputStream();
        BufferedReader reader=
            new BufferedReader(new InputStreamReader(in));
        SOQuestions questions=
            new Gson().fromJson(reader, SOQuestions.class);

        reader.close();

        EventBus.getDefault().post(new QuestionsLoadedEvent(questions));
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
      }
      finally {
        c.disconnect();
      }
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
    }
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

LoadThread:

- Creates an HttpUrlConnection by creating a URL for our Stack Exchange API endpoint and opening a connection
- Creates a BufferedReader wrapped around the InputStream from the HTTP connection
- Parses the JSON we get back from that HTTP request via a Gson instance, loading the data into an instance of our SOQuestions
- Close the BufferedReader (and the InputStream by extension)
- Post a QuestionsLoadedEvent to greenrobot's EventBus, to let somebody know that our questions exist
- Log messages to LogCat in case of errors

QuestionsLoadedEvent is a simple wrapper around an SOQuestions instance, serving as an event class for use with EventBus:

```
package com.commonsware.android.hurl;

public class QuestionsLoadedEvent {
  final SOQuestions questions;

  QuestionsLoadedEvent(SOQuestions questions) {
    this.questions=questions;
  }
}
```

A Fragment for Questions

The sample app has a QuestionsFragment that should display these loaded questions:

```
package com.commonsware.android.hurl;

import android.app.ListFragment;
import android.os.Bundle;
import android.text.Html;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;
import java.util.List;
import de.greenrobot.event.EventBus;

public class QuestionsFragment extends ListFragment {
  @Override
  public void onCreate(Bundle savedInstanceState) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onCreate(savedInstanceState);

    setRetainInstance(true);
    new LoadThread().start();
  }

  @Override
  public void onResume() {
    super.onResume();
    EventBus.getDefault().register(this);
  }

  @Override
  public void onPause() {
    EventBus.getDefault().unregister(this);
    super.onPause();
  }

  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    Item item=((ItemsAdapter)getListAdapter()).getItem(position);

    EventBus.getDefault().post(new QuestionClickedEvent(item));
  }

  public void onEventMainThread(QuestionsLoadedEvent event) {
    setListAdapter(new ItemsAdapter(event.questions.items));
  }

  class ItemsAdapter extends ArrayAdapter<Item> {
    ItemsAdapter(List<Item> items) {
      super(getActivity(), android.R.layout.simple_list_item_1, items);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      View row=super.getView(position, convertView, parent);
      TextView title=(TextView)row.findViewById(android.R.id.text1);

      title.setText(Html.fromHtml(getItem(position).title));

      return(row);
    }
  }
}
```

In `onCreate()`, we mark that this fragment should be retained and fork the `LoadThread`. Hence, once we have our questions, our retained fragment will hold onto that model data for us, and we avoid duplicating the `LoadThread` if a configuration change occurs sometime after our fragment was initially created.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In `onResume()` and `onPause()`, we register and unregister from the `EventBus`. Our `onEventMainThread()` method will be called when the `QuestionsLoadedEvent` is raised by `LoadThread`, and there we hold onto the loaded questions and populate the `ListView`. We use an `ItemsAdapter`, which knows how to render an `Item` as a simple `ListView` row showing the question title. The `ItemsAdapter` uses `Html.fromHtml()` to populate the `ListView` rows, not because Stack Overflow hands back titles with HTML tags, but because Stack Overflow hands back titles with HTML *entity references*, and `Html.fromHtml()` should handle many of those.

And, in `onListItemClick()`, we find the `Item` associated with the row that the user clicked upon, then raise a `QuestionClickedEvent` to let somebody know that the user clicked on that row. The `QuestionClickedEvent` class is a simple wrapper around an `Item` to serve as an event class for use with `EventBus`:

```
package com.commonsware.android.hurl;

public class QuestionClickedEvent {
  final Item item;

  QuestionClickedEvent(Item item) {
    this.item=item;
  }
}
```

An Activity for Orchestration

`MainActivity` sets up the fragment in `onCreate()`, registers and unregisters for the event bus in `onResume()` and `onPause()`, and handles the click events in `onEventMainThread()`:

```
package com.commonsware.android.hurl;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import de.greenrobot.event.EventBus;

public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new QuestionsFragment()).commit();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }

  @Override
  public void onResume() {
    super.onResume();
    EventBus.getDefault().register(this);
  }

  @Override
  public void onPause() {
    EventBus.getDefault().unregister(this);
    super.onPause();
  }

  public void onEventMainThread(QuestionClickedEvent event) {
    startActivity(new Intent(Intent.ACTION_VIEW,
                             Uri.parse(event.item.link)));
  }
}
```

Hence, `MainActivity` is serving in an orchestration role. `QuestionsFragment` is a local controller, handling direct events raised by its widgets (a `ListView`). `MainActivity` is responsible for handling events that transcend an individual fragment — in this case, it starts a browser to view the clicked-upon question.

The result is a simple `ListView` showing questions:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

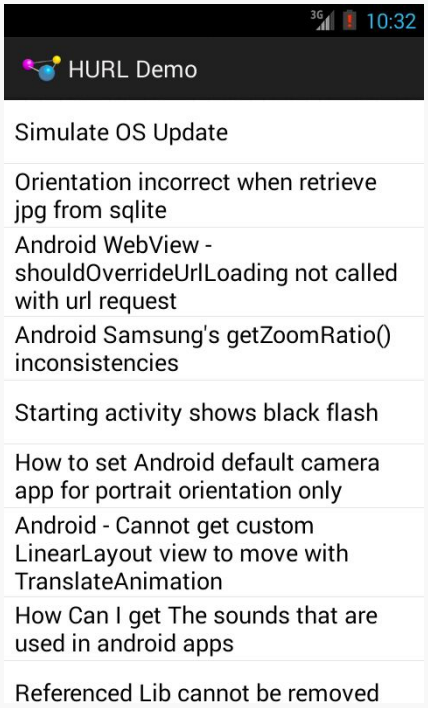


Figure 279: HURLDemo, Showing Stack Overflow Questions

What Android Brings to the Table

Google has augmented `HttpUrlConnection` to do more stuff to help developers. Notably:

- It automatically uses GZip compression on requests, adding the appropriate HTTP header and automatically decompressing any compressed responses (added in Android 2.3)
- It uses Server Name Indication to help work with several HTTPS hosts sharing a single IP address
- API Level 13 (Android 4.0) added an `HttpResponseCache` implementation of the `java.net.ResponseCache` base class, that can be installed to offer transparent caching of your HTTP requests.

Also, courtesy of some third-party code (OkHttp) that we will discuss shortly, `HttpUrlConnection` also supports the SPDY protocol for accelerating Web content distribution over SSL as of Android 4.4.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Testing with StrictMode

`StrictMode`, mentioned in the chapter on files, can also report on performing network I/O on the main application thread. More importantly, on Android 4.0 and higher, the emulator will, by default, crash your app if you try to perform network I/O on the main application thread.

Hence, it is generally a good idea to test your app, either using `StrictMode` yourself or using a suitable emulator, to make sure that you are not performing network I/O on the main application thread.

What About HttpClient?

Android also contains a mostly-complete copy of version 4.0.2beta of the Apache HttpClient library. Many developers use this, as they prefer the richer API offered by this library over the somewhat more clunky approach used by `java.net`. And, truth be told, this was the more stable option prior to Android 2.3.

There are a few reasons why this is no longer recommended, for Android 2.3 and beyond:

- The core Android team is better able to add capabilities to the `java.net` implementation while maintaining backwards compatibility, because its API is more narrow.
- The problems previously experienced on Android with the `java.net` implementation have largely been fixed.
- The Apache HttpClient project continuously evolves its API. This means that Android will continue to fall further and further behind the latest-and-greatest from Apache, as Android insists on maintaining the best possible backwards compatibility and therefore cannot take on newer-but-different HttpClient versions.

That being said, you are welcome to use HttpClient if you are not concerned about these limitations.

HTTP via DownloadManager

If your objective is to download some large file, you may be better served by using the `DownloadManager` added to Android 2.3, as it handles a lot of low-level complexities for you. For example, if you start a download on WiFi, and the user

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

leaves the building and the device fails over to some form of mobile data, you need to reconnect to the server and either start the download again or use some content negotiation to pick up from where you left off. `DownloadManager` handles that.

However, `DownloadManager` is dependent upon some broadcast `Intent` objects, a technique we have not discussed yet, so we will delay covering `DownloadManager` until later in the book.

Using Third-Party JARs

To some extent, the best answer is to not write the code yourself, but rather use some existing JAR that handles both the Internet I/O and any required data parsing. This is commonplace when accessing public Web services — either because the firm behind the Web service has released a JAR, or because somebody in the community has released a JAR for that Web service.

Examples include:

- Using JTwitter to access Twitter's API
- Using Amazon's JAR to access various AWS APIs, including S3, SimpleDB, and SQS
- Using the Dropbox SDK for accessing DropBox folders and files

However, beyond the classic potential JAR problems, you may encounter another when it comes to using JARs for accessing Internet services: versioning. For example:

- JTwitter bundles the `org.json` classes in its JAR, which will be superseded by Android's own copy, and if the JTwitter version of the classes have a different API, JTwitter could crash.
- Libraries dependent upon HttpClient might be dependent upon a version with a different API (e.g., 4.1.1) than is in Android (4.0.2 beta).

Try to find JARs that have been tested on Android and are clearly supported as such by their author. Lacking that, try to find JARs that are open source, so you can tweak their implementation if needed to add Android support.

Later in this chapter, we will review another class of third-party JARs, ones that are more general-purpose than things like JTwitter, but still offer to simplify HTTP processing.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

SSL

Of course, if you are thinking about HTTP, you really should be thinking about HTTPS — SSL-encrypted HTTP operations.

Normally, SSL "just works", by using an `https://` URL. Hence, typically, there is little that you need to do to enable simple encryption.

However, there are other aspects of SSL to consider, including:

- What if the server is not using an SSL certificate that Android will honor, such as a self-signed certificate?
- What about man-in-the-middle attacks, hacked certificate authorities, and the like?

The trails contain a chapter dedicated to SSL that you are encouraged to read, so that this chapter does not get crazy-long.

Using HTTP Client Libraries

Often times, writing Internet access code is a pain in various body parts.

Not surprisingly, there are a variety of third-party libraries designed to assist with this. Some are designed to provide access to a specific API, such as the ones mentioned earlier in this chapter. However, others are more general-purpose, designed to make writing HTTP operations a bit easier, by handling things like:

- Retries (e.g., device failed over from WiFi to mobile data mid-transaction)
- Threading (e.g., handling doing the Internet work on a background thread for you)
- Data parsing and marshaling, for well-known formats (e.g., JSON)

In this section, we will look at four libraries that exemplify this approach: OkHttp, Retrofit, Picasso, and Ion. Later, we will see other libraries that you might wish to investigate, including Google's own Volley HTTP client API.

OkHttp

OkHttp uses a modified clone of the standard `HttpUrlConnection` to offer many performance improvements. Most notable is its support for SPDY, a Google

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

sponsored enhanced version of HTTP, going beyond classic HTTP "keep-alive" support to allow for many requests and responses to be delivered over the same socket connection. Many Google APIs are served by SPDY-capable servers, and SPDY support is available for others to use as well.

Beyond that, OkHttp wraps up common HTTP performance-improvement patterns, such as GZIP compression, response caching, and connection pooling. It also is more aware of "real world" connection issues, like mis-configured proxy servers and the like.

OkHttp supports both `HttpUrlConnection`- and `HttpClient`-compatible APIs. To accommodate this, OkHttp is broken up into three JARs:

- One for OkHttp's core (`okhttp-X.Y.Z.jar`)
- One for the `HttpUrlConnection` API (`okhttp-urlconnection-X.Y.Z.jar`)
- One for the `HttpClient` API (`okhttp-apache-X.Y.Z.jar`)

Android Studio users can add a dependency on either `com.squareup.okhttp:okhttp-urlconnection:...` or `com.squareup.okhttp:okhttp-apache:...`, for some version denoted by `...`. This will automatically pull down the OkHttp core code as well.

You will need to choose which of those you want. Going with the core JAR plus one of the API JARs means that your existing code that uses the classic APIs will require very little in the way of change. For example, to use the `HttpUrlConnection` API, given the requisite two JARs, is mostly a matter of creating an instance of `OkHttpClient` and using its `open()` method to open a connection to a `URL`. `open()` will return an `HttpUrlConnection`, which you use the same as you would normally, except that OkHttp's implementation adds the aforementioned features.

However, you are still stuck with the `HttpUrlConnection` or `HttpClient` APIs, handling things like request generation and response processing yourself. Square's other two HTTP client libraries — Retrofit and Picasso — layer atop of OkHttp to help address usability issues for specific scenarios. And OkHttp 2.0 added a new native API that is designed to be simpler and cleaner than either `HttpUrlConnection` or `HttpClient`.

Note that OkHttp is used as the standard implementation of `HttpUrlConnection` in Android 4.4 and higher — this is where Android's SPDY support comes from.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Retrofit

Many times, when working with HTTP requests, our needs are fairly simple: just retrieve some JSON (or other structured data, such as XML) from some Web service, or perhaps upload some JSON to that Web service.

Retrofit is designed to simplify this, by handling the data parsing and marshaling for us, along with the HTTP operations and (optionally) background threading. We are left with a fairly natural-looking Java API to send/receive Java objects to/from the Web service. Retrofit accomplishes this through the cunning use of annotations, reflection, and, where available, OkHttp itself.

To demonstrate Retrofit, let's review the `HTTP/Retrofit` sample project. This project is a clone of the preceding one, this time using Retrofit for retrieving and parsing our Stack Overflow questions.

Note that Stack Overflow happens to use JSON as its data format, which works nicely with Retrofit, as JSON is its default data format. However, you can supply your own conversion logic, to convert data to/from other formats, such as XML or Protocol Buffers.

Downloading and Installing Retrofit

Retrofit is available as a small JAR from the aforementioned Retrofit Web site. By default, it uses Google's Gson for its JSON parsing, so you will need to download that JAR as well. If you also have OkHttp in your project, Retrofit will automatically use it, falling back to standard Android HTTP APIs if OkHttp is unavailable.

The combination of these JARs totals around 600KB, mostly coming from Gson and OkHttp. For most apps, this will not be a major issue, but if APK size is important to you, you need to keep in mind the footprint that these JARs consume.

Creating Your Service Interface

The first thing we need is to tell Retrofit more about where our JSON is coming from. To do this, we need to create a Java interface with some specific Retrofit-supplied annotations, documenting:

- the HTTP operations that we wish to perform
- the path (and, if needed, query parameters) to apply an HTTP operation to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- the per-request data to configure the HTTP operation, such as the dynamic portions of the path for a REST-style API, or additional query parameters to attach to the URL
- what object should be used for pouring the HTTP response into

For example, let's take a look at `StackOverflowInterface`, our interface for making a query of Stack Exchange's API to get questions from Stack Overflow:

```
package com.commonsware.android.retrofit;

import retrofit.Callback;
import retrofit.http.GET;
import retrofit.http.Query;

public interface StackOverflowInterface {
  @GET("/2.1/questions?order=desc&sort=creation&site=stackoverflow")
  void questions(@Query("tagged") String tags, Callback<SOQuestions> cb);
}
```

Each method in the interface should have an annotation identifying the HTTP operation to perform, such as `@GET` or `@POST`. The parameter to the annotation is the path for the request and any fixed query parameters. In our case, we are using the path documented by Stack Exchange for retrieving questions (`/2.1/questions`), plus some fixed query parameters:

- `order` for whether the results should be ascending (`asc`) or descending (`desc`)
- `sort` to indicate how the questions should be sorted, such as `creation` to sort by time when the question was posted
- `site` to indicate what Stack Exchange site we are querying (e.g., `stackoverflow`)

The method name can be whatever you want.

If you have additional query parameters that vary dynamically, you can use the `@Query` annotation on `String` parameters to have them be added to the end of the URL. In our case, the `tagged` query parameter will be added with whatever the `tags` parameter is to our `questions()` method.

Similarly, you can use `{name}` placeholders for path segments, and replace those at runtime via `@Path`-annotated parameters to the method.

To get results back, and indicate the data type for those results, you have two choices:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. Have the method return the data type you wish, in which case when we eventually call this method, the HTTP operation will be performed synchronously, blocking our method call
2. Pass a `Callback` parameter, declared with the desired data type (e.g., `SOQuestions`), in which case when we eventually call this method, the HTTP operation will be performed on a background thread, with the results delivered to us asynchronously via a custom `Callback` implementation that we will supply

In this case, we are electing to let Retrofit handle the threading for us, so we supply a `Callback` and have the method return `void`.

Curiously, we will never create an implementation of the `StackOverflowInterface` ourselves. Instead, Retrofit generates one for us, with code that implements our requested behaviors.

Creating the RestAdapter

To use this generated `StackOverflowInterface`, and to actually perform these operations, we need to create an instance of a `RestAdapter`. Usually, you will do this via a `RestAdapter.Builder`, to configure what you want done.

The biggest thing you will provide to `RestAdapter.Builder` is the server tied to these HTTP operations. Calling `setEndpoint()` allows you to specify the scheme, host, and port to be attached to the rest of the URL, coming from your interface. For example, we need to make our requests of the `https://api.stackexchange.com` server, so we have:

```
    RestAdapter restAdapter=
        new RestAdapter.Builder().setEndpoint("https://api.stackexchange.com")
                                 .build();
```

Other methods on `RestAdapter.Builder` include:

- `setConverter()`, if your payloads are not in JSON format, but something else
- `setExecutors()`, to provide `Executor` objects (e.g., instances of `ThreadPoolExecutor`) to be used for requests and callbacks
- `setLog()` and `setDebug()` for controlling log output

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

When you are done configuring the `RestAdapter.Builder`, call `build()` to get the resulting `RestAdapter`.

Making Requests

Given a configured `RestAdapter`, you can retrieve an implementation of your API interface by calling the `create()` method:

```
    StackOverflowInterface so=
        restAdapter.create(StackOverflowInterface.class);
```

You can then use the resulting interface-typed object no differently than you would any other Java object, despite the fact that you never wrote an implementation of that interface yourself.

In our case, we can call the `questions()` method, supplying the tag (or tags) from which we wish to receive recent questions:

```
    so.questions("android", this);
```

The second parameter to `questions()` is an implementation of `Callback`, to receive asynchronous results from our HTTP `GET` request. `Callback` requires two methods, `success()` and `failure()`. `success()` takes two parameters: the data type you indicated in the interface (e.g., `SOQuestions`) representing the parsed results of the HTTP request, and a `Response` object containing other information from the HTTP response, such as headers:

```
  @Override
  public void success(SOQuestions questions, Response response) {
    setListAdapter(new ItemsAdapter(questions.items));
  }
```

Here, we update our `ListView` with an `ItemsAdapter` based upon the received questions.

`failure()` takes a single parameter, an instance of `RetrofitError`, which is an `Exception` providing details of something that went wrong in the HTTP request (e.g., authorization was denied). You can handle that no differently than you might other exceptions from elsewhere in your app, to let the user know that something went wrong. In this case, we take the crude-but-easy approach of showing a `Toast` and logging the details to LogCat:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@Override
public void failure(RetrofitError exception) {
  Toast.makeText(getActivity(), exception.getMessage(),
                 Toast.LENGTH_LONG).show();
  Log.e(getClass().getSimpleName(),
        "Exception from Retrofit request to StackOverflow", exception);
}
```

Picasso

Sometimes, what you want to download is not JSON, or XML, or any sort of structured data.

Sometimes, it is an image.

For example, Stack Overflow users have avatars. In our sample app, it might be nice to display the avatar of the user who asked the question.

Picasso is a library from Square that is designed to help with asynchronously loading images, whether those images come from HTTP requests, local files, a `ContentProvider`, etc. In addition to doing the loading asynchronously, Picasso simplifies many operations on those images, such as:

- Caching the results in memory (or optionally on disk for HTTP requests)
- Displaying placeholder images while the real images are being loaded, and displaying error images if there was a problem in loading the image (e.g., invalid URL)
- Transforming the image, such as resizing or cropping it to fit a certain amount of space
- Loading the images directly into an `ImageView` of your choice, even handling cases where that `ImageView` is recycled (e.g., part of a row in a `ListView`, where the user scrolled while an image for that `ImageView` was still loading, and now *another* image is destined for that same `ImageView` when the row was recycled)

The `HTTP/Picasso` sample application extends the Retrofit one to download the avatar image of the person asking the question, displaying it in the `ListView` along with the question title.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Downloading and Installing Picasso

Picasso can be downloaded as a small JAR from the aforementioned Web site. It requires OkHttp 1.6.0+, both the core JAR and the OkHttp `HttpUrlConnection` JAR. Android Studio users can just add a dependency for `com.squareup.picasso:picasso:...` for some version denoted by `...`, and it will pull down all other dependencies needed by Picasso.

Updating the Model

Our original data model did not include information about the owner. Hence, we need to augment our data model, so Retrofit pulls that information out of the Stack Exchange JSON and makes it available to us.

To that end, we now have an `Owner` class, holding onto the one piece of information we need about the owner: the URL to the avatar (a.k.a., "profile image"):

```
package com.commonsware.android.picasso;

import com.google.gson.annotations.SerializedName;

public class Owner {
  @SerializedName("profile_image") String profileImage;
}
```

The JSON key for this in the Stack Exchange API is `profile_image`, and underscores are not the conventional way of separating words in a Java data member. Java samples usually use "camelCase" instead. The default behavior of Retrofit would require us to name our data member `profile_image` to match the JSON.

However, under the covers, Retrofit is using Google's Gson to do the mapping from JSON to objects. Gson supports a `@SerializedName` annotation, to indicate the JSON key to use for this data member. This allows us to give the data member the more natural name of `profileImage`, by using `@SerializedName("profile_image")` to teach Gson how to populate it properly.

(The author would like to thank Alec Holmes for his assistance with the Gson support)

Our `Item` class now has an `Owner`, named `owner`, since the owner data is in the `owner` key of an item's JSON object:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.picasso;

public class Item {
  String title;
  Owner owner;
  String link;

  @Override
  public String toString() {
    return(title);
  }
}
```

Those two changes are sufficient for Retrofit to give us our URL to be able to download the image.

Requesting the Images

Using Picasso is extremely simple, as it offers a fluent interface that allows us to set up a request in a single Java statement.

The statement begins with a call to the static `with()` method on the `Picasso` class, where we supply a `Context` (such as our activity) for Picasso to use. The statement *ends* with a call to `into()`, indicating the `ImageView` into which Picasso should load an image. In between those calls, we can chain other calls, as `with()` and most other methods on a `Picasso` object return the `Picasso` object itself.

So, we can do something like:

```
      Picasso.with(getActivity()).load(item.owner.profileImage)
             .fit().centerCrop()
             .placeholder(R.drawable.owner_placeholder)
             .error(R.drawable.owner_error).into(icon);
```

Here, we:

- Indicate that we want to `load()` an image found at a certain URL, identified by the `profileImage` data member of the `Owner` inside an `Item` referred to as `item`
- Say that we want to `fit()` the image to our target `ImageView`
- Specify that the image should be resized using `centerCrop()` rules, to center the image within the desired size (if it is smaller on one or both axes) and to crop the image (if it is larger on one or both axes)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Indicate that we want to put a certain drawable resource as the `placeholder()` image to show in the `ImageView` while the loading is going on in the background
- State that we want to show a certain drawable resource in the `ImageView` in case of an `error()` when the image was being loaded

And that's it. Picasso will go off, download the image, and pour it into the `ImageView` when it is ready (and resized).

The Rest of the Story

That bit of Picasso code is in a new `getView()` method on our `ItemsAdapter`:

```
  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    View row=super.getView(position, convertView, parent);
    Item item=getItem(position);
    ImageView icon=(ImageView)row.findViewById(R.id.icon);

    Picasso.with(getActivity()).load(item.owner.profileImage)
           .fit().centerCrop()
           .placeholder(R.drawable.owner_placeholder)
           .error(R.drawable.owner_error).into(icon);

    TextView title=(TextView)row.findViewById(R.id.title);

    title.setText(Html.fromHtml(getItem(position).title));

    return(row);
  }
```

We have created our own row layout (`res/layout/row.xml`), consisting of an `ImageView` and a `TextView`. We have `ArrayAdapter` inflate or recycle our row, retrieve the `Item` for this row, retrieve the `ImageView` out of the row, use Picasso to start loading the real image, fill in the HTML-entity-aware text into the `TextView`, and then return our updated row. By the time we return the row, Picasso will have already loaded the placeholder image, which is what the user will initially see, while we download the real image.

The result is that we now have icons next to each of our question titles:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

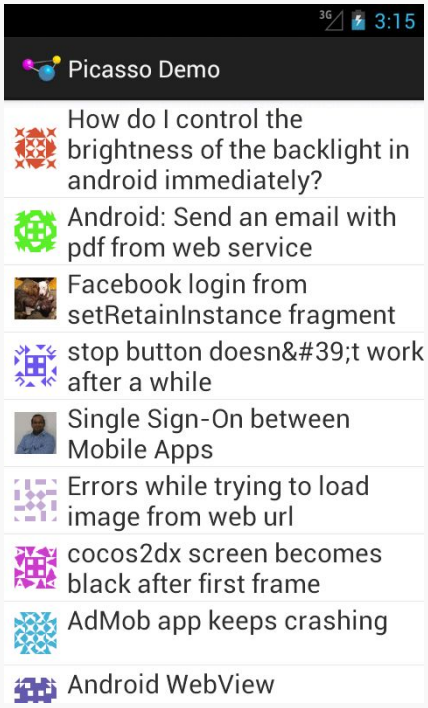


Figure 280: The Picasso Demo App

Ion

One complaint regarding the combination of Retrofit and Picasso is that you need two libraries. Picasso knows images, but cannot perform non-image HTTP requests, and Retrofit knows REST but not images.

Ion is a library that covers both, so let's see how it can be used, via the `HTTP/Ion` sample application. This is a port of the Retrofit/Picasso sample app, replacing those libraries with Ion.

Getting the Questions

In the revised `QuestionsFragment`, in `onCreateView()`, we use Ion to retrieve the questions as a `JsonObject`:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=
        super.onCreateView(inflater, container, savedInstanceState);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    setRetainInstance(true);

    Ion.with(getActivity(),
             "https://api.stackexchange.com/2.1/questions?"
                 + "order=desc&sort=creation&site=stackoverflow&"
                 + "tagged=android").asJsonObject().setCallback(this);

    return(result);
```

Ion, like Picasso, uses a fluent, builder-style interface. The static `with()` method on the `Ion` class takes a `Context` and the URL (or `Uri`) of the data to be retrieved. We then cascade onto the return value:

- a call to `asJsonObject()`, indicating that we want Ion to convert the JSON into a Gson `JsonObject`
- a call to `setCallback()`, indicating where the `JsonObject` should be delivered

The `setCallback()` call takes a `FutureCallback` as a parameter; we implement that interface on the `QuestionsFragment` itself. That, in turn requires that we implement `onCompleted()`:

```
  @Override
  public void onCompleted(Exception e, JsonObject json) {
    if (e != null) {
      Toast.makeText(getActivity(), e.getMessage(), Toast.LENGTH_LONG)
           .show();
      Log.e(getClass().getSimpleName(),
            "Exception from Retrofit request to StackOverflow", e);
    }

    if (json != null) {
      JsonArray items=json.getAsJsonArray("items");
      ArrayList<JsonObject> normalized=new ArrayList<JsonObject>();

      for (int i=0; i < items.size(); i++) {
        normalized.add(items.get(i).getAsJsonObject());
      }

      setListAdapter(new ItemsAdapter(normalized));
    }
  }
```

We are passed an `Exception` if there was a problem, or the `JsonObject` if we did indeed receive results. If we have an `Exception`, we show the message via a `Toast` and log it to LogCat. If we have a `JsonObject`, we convert the `JsonArray` of items into an `ArrayList` of `JsonObjects`, so that we can use `ArrayAdapter` (which

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

understands `java.util.List` but not `JsonArray`, as the Gson authors did not implement the `List` interface on `JsonArray`).

Getting the Avatars

That means that `ItemsAdapter` is now an `ArrayAdapter` of `JsonObject`, rather than an `ArrayAdapter` of a custom `Item` class. Our `getView()` method therefore must also be updated, both to grab the title and avatar URL from the `JsonObject` representing a question, and to use Ion to retrieve the avatar image:

```
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      View row=super.getView(position, convertView, parent);
      JsonObject item=getItem(position);
      ImageView icon=(ImageView)row.findViewById(R.id.icon);

      Ion.with(icon)
         .placeholder(R.drawable.owner_placeholder)
         .resize(size, size)
         .centerCrop()
         .error(R.drawable.owner_error)
         .load(item.getAsJsonObject("owner").get("profile_image")
                   .getAsString());

      TextView title=(TextView)row.findViewById(R.id.title);

      title.setText(Html.fromHtml(item.get("title").getAsString()));

      return(row);
    }
```

With Gson, `get()` on a `JsonObject` returns a `JsonElement`, which in turn can be coerced into various data types, such as using `getAsString()` to coerce it into a `String`. For example, we use that when populating the `TextView` with the question's title.

To retrieve the avatar and populate the `ImageView` with it, we use Ion much in the same fashion as we used Picasso. We use a different version of the static `with()` method, one that takes our `ImageView` as a parameter. The rest of the fluent API is similar to Picasso's:

- `placeholder()` and `error()` indicate drawables to use when the image is loading or if an HTTP error was encountered, respectively
- `resize()` and `centerCrop()` allow us to crop the image to the desired size

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `load()` supplies the URL from which to retrieve the image, here obtained by driving down into Gson data structures to get the `profile_image` URL of the `owner` of this question

Visually, the result is the same as the Picasso example.

Other Candidate Libraries

There are plenty of other libraries that similarly try to help simplify Android HTTP operations, including:

- android-async-http
- android-json-rpc
- Android-Universal-Image-Loader
- basic-http-client
- http-request
- ImageLoader
- OpenS
- Smart Image View

and countless others.

Hey, What About Volley?

At the Google I|O 2013 conference, there was a session about Volley, an HTTP client library created by Google and used by internal apps, such as the Play Store. Volley can be thought of as a superset of Retrofit plus Picasso, minus Picasso's non-HTTP image loading facilities.

On the plus side, Volley *is* such a superset and therefore a single code base can be used to replace multiple libraries. Also, given Volley's use by Google, one imagines that this code has been applied to the widest range of possible devices.

However, Volley is distributed as just a dump of source code. There is no packaging of the code into a JAR. There is no documentation beyond that I|O video and a training module. There is no support mechanism, except perhaps via ad-hoc social media inquiries and general support sites (e.g., Stack Overflow).

If you are a fairly expert developer, and wish to experiment with Volley, there are plenty of others with a similar interest, and perhaps the community will build up its own knowledge base and be able to support Volley users. Otherwise, you may be

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

better served sticking with libraries that have more packaging, documentation, and support structures.

Visit the Trails

As noted earlier, there is a chapter on SSL that you should read, if you run into trouble using SSL in Android or want to improve your security further than you get with just stock SSL handling.

There is also a chapter on miscellaneous network capabilities – the coverage of `DownloadManager` can be found there.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Intents, Intent Filters

We have seen `Intent` objects briefly, in our discussion of having multiple activities in our application. However, we really did not dive into too much of the details about those `Intent` objects, and they can be used in other ways besides starting up an activity. In this chapter, we will examine `Intent` and their filters.

What's Your Intent?

When Sir Tim Berners-Lee cooked up the Hypertext Transfer Protocol — HTTP – he set up a system of verbs plus addresses in the form of URLs. The address indicated a resource, such as a Web page, graphic, or server-side program. The verb indicated what should be done: GET to retrieve it, POST to send form data to it for processing, etc.

An `Intent` is similar, in that it represents an action plus context. There are more actions and more components to the context with `Intent` than there are with HTTP verbs and resources, but the concept is still the same.

Just as a Web browser knows how to process a verb+URL pair, Android knows how to find activities or other application logic that will handle a given `Intent`.

Pieces of Intents

The two most important pieces of an `Intent` are the action and what Android refers to as the "data". These are almost exactly analogous to HTTP verbs and URLs — the action is the verb, and the "data" is a `Uri`, such as `http://commonsware.com` representing an HTTP URL to some balding guy's Web site. Actions are constants, such as `ACTION_VIEW` (to bring up a viewer for the resource) or `ACTION_EDIT` (to edit the resource).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you were to create an `Intent` combining `ACTION_VIEW` with a content `Uri` of `http://commonsware.com`, and pass that `Intent` to Android via `startActivity()`, Android would know to find and open an activity capable of viewing that resource.

There are other criteria you can place inside an `Intent`, besides the action and "data" `Uri`, such as:

1. Categories. Your "main" activity will be in the `LAUNCHER` category, indicating it should show up on the launcher menu. Other activities will probably be in the `DEFAULT` category, though other categories exist and are used on occasion.
2. A MIME type, indicating the type of resource you want to operate on.
3. A component, which is to say, the class of the activity that is supposed to receive this `Intent`.
4. "Extras", which is a `Bundle` of other information you want to pass along to the receiver with the `Intent`, that the recipient might want to take advantage of. What pieces of information a given recipient can use is up to the recipient and (hopefully) is well-documented.

You will find rosters of the standard actions, categories, and extras in the Android SDK documentation for the `Intent` class.

Intent Routing

As noted above, if you specify the target component in your `Intent`, Android has no doubt where the `Intent` is supposed to be routed to — it will launch the named activity. This might be OK if the target recipient (e.g., the activity to be started) is in your application. It definitely is not recommended for invoking functionality in other applications. Component names, by and large, are considered private to the application and are subject to change. Actions, `Uri` templates, and MIME types are the preferred ways of identifying capabilities you wish third-party code to supply.

If you do not specify the target component, then Android has to figure out what recipients are eligible to receive the `Intent`. For example, Android will take the `Intent` you supply to `startActivity()` and find the activities that might support it. Note the use of the plural "activities", as a broadly-written intent might well resolve to several activities. That is the... ummm... intent (pardon the pun), as you will see later in this chapter. This routing approach is referred to as implicit routing.

Basically, there are three rules, all of which must be true for a given activity to be eligible for a given `Intent`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- The activity must support the specified action
- The activity must support the stated MIME type (if supplied)
- The activity must support all of the categories named in the `Intent`

The upshot is that you want to make your `Intent` specific enough to find the right recipient, and no more specific than that.

This will become clearer as we work through some examples throughout this chapter.

Stating Your Intent(ions)

All Android components that wish to be started via an `Intent` must declare `Intent` filters, so Android knows which intents should go to that component. A common approach for this is to add one or more `<intent-filter>` elements to your `AndroidManifest.xml` file, inside the element for the component that should respond to the `Intent`.

For example, all of the sample projects in this book have an `<intent-filter>` on an `<activity>` that looks like this:

```
<intent-filter>
  <action android:name="android.intent.action.MAIN"/>
  <category android:name="android.intent.category.LAUNCHER"/>
</intent-filter>
```

Here, we declare that this activity:

1. Is the main activity for this application
2. It is in the `LAUNCHER` category, meaning it gets an icon in anything that thinks of itself as a "launcher", such as the home screen

You are welcome to have more than one action or more than one category in your `Intent` filters. That indicates that the associated component (e.g., activity) handles multiple different sorts of `Intent` patterns.

Responding to Implicit Intents

We saw in the chapter on multiple activities how one activity can start another via an explicit `Intent`, identifying the particular activity to be started:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
startActivity(new Intent(this, OtherActivity.class));
```

In that case, `OtherActivity` does not need an `<intent-filter>` in the manifest. It will automatically respond when somebody explicitly identifies it as the desired activity.

However, what if you want to respond to an implicit `Intent`, one that focuses on an action string and other values? Then you *will* need an `<intent-filter>` in the manifest.

For example, take a look at the `Intents/FauxSender` sample project.

Here, we have an activity, `FauxSender`, set up to respond to an `ACTION_SEND` `Intent`, specifically for content that has the MIME type of `text/plain`:

```
    <activity
      android:name="FauxSender"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter android:label="@string/app_name">
        <action android:name="android.intent.action.SEND"/>

        <data android:mimeType="text/plain"/>

        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
    </activity>
```

The call to `startActivity()` will always add the `DEFAULT` category if no other category is specified, which is why our `<intent-filter>` also filters on that category.

Hence, if somebody on the system calls `startActivity()` on an `ACTION_SEND` `Intent` with a MIME type of `text/plain`, our `FauxSender` activity might get control. We will explain the use of the term "might" in the next section.

The documentation for `ACTION_SEND` indicates that a standard extra on the `Intent` is `EXTRA_TEXT`, representing the text to be sent. There might also be an `EXTRA_SUBJECT`, representing a subject line, if the "send" operation might have such a concept, such as an email client.

`FauxSender` can retrieve those extras and make use of them:

```
package com.commonsware.android.fsender;

import android.app.Activity;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.content.Intent;
import android.os.Bundle;
import android.text.TextUtils;
import android.widget.Toast;

public class FauxSender extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    String msg=getIntent().getStringExtra(Intent.EXTRA_TEXT);

    if (TextUtils.isEmpty(msg)) {
      msg=getIntent().getStringExtra(Intent.EXTRA_SUBJECT);
    }

    if (TextUtils.isEmpty(msg)) {
      Toast.makeText(this, R.string.no_message_supplied,
                     Toast.LENGTH_LONG).show();
    }
    else {
      Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
    }

    finish();
  }
}
```

Here, we use `TextUtils.isEmpty()` to detect if an extra is either `null` or has an empty string as its value. If `EXTRA_TEXT` is supplied, we show it in a `Toast`. Otherwise, we use `EXTRA_SUBJECT` if it is supplied, and if that is also missing, we show a stock message from a string resource.

The activity then immediately calls `finish()` from `onCreate()` to get rid of itself. That, coupled with `android:theme="@android:style/Theme.NoDisplay"` in the `<activity>` element, means that the activity will have no user interface, beyond the `Toast`. If run from the launcher, you will still see the launcher behind the `Toast`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

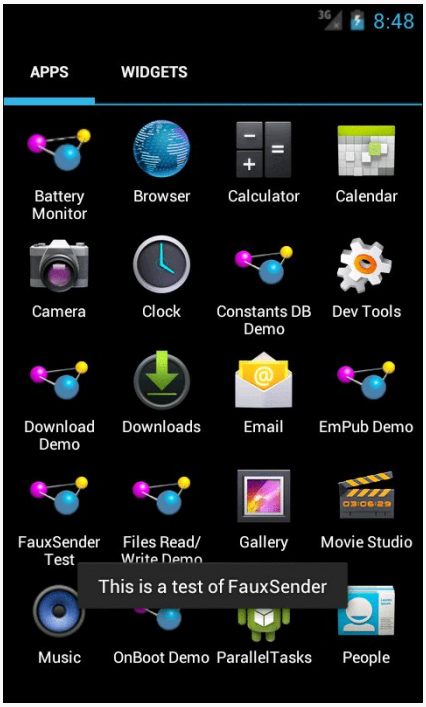


Figure 281: FauxSender, Showing EXTRA_TEXT

Requesting Implicit Intents

To send something via `ACTION_SEND`, you first set up the `Intent`, containing whatever information you want to send in `EXTRA_TEXT`, such as this code from the `FauxSenderTest` activity:

```
    Intent i=new Intent(Intent.ACTION_SEND);

    i.setType("text/plain");
    i.putExtra(Intent.EXTRA_SUBJECT, R.string.share_subject);
    i.putExtra(Intent.EXTRA_TEXT, theMessage);
```

(where `theMessage` is a passed-in parameter to the method containing this code fragment)

If we call `startActivity()` on this `Intent` directly, there are three possible outcomes, described in the following sections.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Zero Matches

It is possible, though unlikely, that there are no activities at all on the device that will be able to handle this `Intent`. In that case, we crash with an `ActivityNotFoundException`. This is a `RuntimeException`, which is why we do not have to keep wrapping all our `startActivity()` calls in `try/catch` blocks. However, if we might start something that does not exist, we really should catch that exception... or avoid the call in the first place. Detecting up front whether there will be any matches for our activity is a topic that will be discussed later in this book.

Note that the odds of an `ActivityNotFoundException` climb substantially on Android 4.3+ tablets, when a restricted profile is in use, as will be discussed later in this book.

One Match

It is possible that there will be exactly one matching activity. In that case, the activity in question starts up and takes over the foreground. This is what we see with the explicit `Intent`.

Many Matches, Default Behavior

It is possible that there will be more than one matching activity. In that case, by default, the user will be presented with a so-called “chooser” dialog box:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

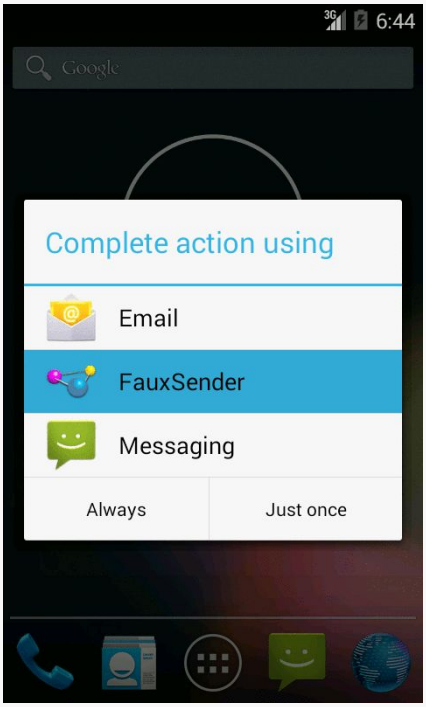


Figure 282: Chooser Dialog

The user can tap on any item in the list to have that particular activity be the one to process this event. And, if the user clicks on "Always", and we invoke the same basic `Intent` again (same action, same MIME type, same categories, same `Uri` scheme), whatever the user chooses *now* will be used again automatically, bypassing the chooser. The "Always" button in the chooser dialog sets the default activity for handling the particular `Intent` structure that triggered the chooser.

The Chooser Override

For many `Intent` patterns, the notion of the user choosing a default makes perfect sense. For example, if the user installs another Web browser, until they set a default activity, every time they go to view a Web page, they will be presented with a chooser, to choose among the installed browsers. This can get annoying quickly.

However, `ACTION_SEND` is one of those cases where a default activity is usually inappropriate. Just because the user on Monday chose to send something via Bluetooth and accidentally clicked "Always" does not mean that every day thereafter, they *always* want *every* `ACTION_SEND` to go via Bluetooth, instead of Gmail or Email or Facebook or Twitter or any other `ACTION_SEND`-capable apps they may have installed.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You can elect to force a chooser to display, regardless of whether the user has set a default activity or not. To do this, instead of calling `startActivity()` on the `Intent` directly, you wrap the `Intent` in another `Intent` returned by the `createChooser()` static method on `Intent` itself:

```
void sendIt(String theMessage) {
  Intent i=new Intent(Intent.ACTION_SEND);

  i.setType("text/plain");
  i.putExtra(Intent.EXTRA_SUBJECT, R.string.share_subject);
  i.putExtra(Intent.EXTRA_TEXT, theMessage);

  startActivity(Intent.createChooser(i,
                                     getString(R.string.share_title)));
}
```

The second parameter to `createChooser()` is a message to appear at the top of the dialog box:

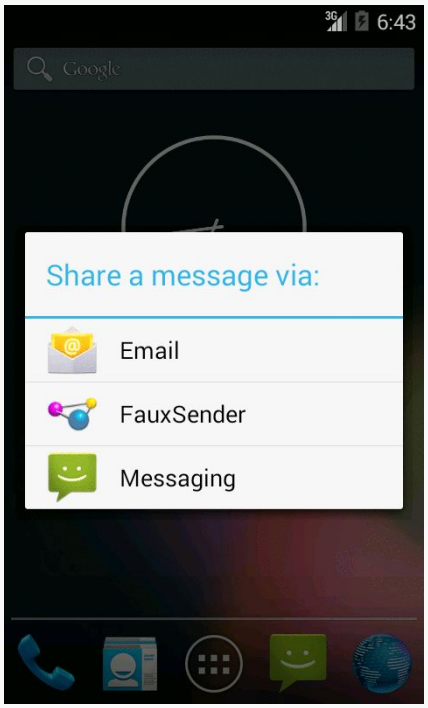


Figure 283: Your Tailored Chooser Dialog

Notice the lack of the "Always" button — not only must the user make a choice now, but also they cannot make a default choice for the future, either.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

ShareActionProvider

Above, we saw how you can bring up a chooser when using `startActivity()` on an implicit `Intent` action, such as `ACTION_SEND`.

There is another option, if you are using the action bar: `ShareActionProvider`. Designed for use with `ACTION_SEND`, `ShareActionProvider` supplies a drop-down menu in the action bar to let the user invoke some implementation of an `Intent` that you configure and supply.

To see how you can add a `ShareActionProvider` to your activity or fragment, let us take a look at the `ActionBar/ShareNative` sample project.

Our activity — `MainActivity` — will utilize the action bar. Its action bar items are contained in a `res/menu/actions.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/share"
    android:actionProviderClass="android.widget.ShareActionProvider"
    android:showAsAction="ifRoom"/>

</menu>
```

In addition to specifying an ID and indicating that the item should always be shown in the action bar, we also include the `android:actionProviderClass` attribute. This points to a concrete implementation of the `ActionProvider` abstract base class, which is responsible for rendering the action bar item. In our case, we are using `ShareActionProvider`.

Our activity UI is simply a large `EditText` widget:

```
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/editor"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:gravity="left|top"
  android:inputType="textMultiLine"/>
```

We load that layout in `onCreate()` of `MainActivity`, along with initializing an `Intent` to be used when we employ the `ShareActionProvider`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.sap;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.text.Editable;
import android.text.TextWatcher;
import android.view.Menu;
import android.widget.EditText;
import android.widget.ShareActionProvider;
import android.widget.Toast;

public class MainActivity extends Activity implements
    ShareActionProvider.OnShareTargetSelectedListener, TextWatcher {
  private ShareActionProvider share=null;
  private Intent shareIntent=new Intent(Intent.ACTION_SEND);
  private EditText editor=null;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.activity_main);

    shareIntent.setType("text/plain");
    editor=(EditText)findViewById(R.id.editor);
    editor.addTextChangedListener(this);
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);

    share=
        (ShareActionProvider)menu.findItem(R.id.share)
                                 .getActionProvider();
    share.setOnShareTargetSelectedListener(this);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onShareTargetSelected(ShareActionProvider source,
                                       Intent intent) {
    Toast.makeText(this, intent.getComponent().toString(),
                   Toast.LENGTH_LONG).show();

    return(false);
  }

  @Override
  public void afterTextChanged(Editable s) {
    shareIntent.putExtra(Intent.EXTRA_TEXT, s.toString());
    share.setShareIntent(shareIntent);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void beforeTextChanged(CharSequence s, int start, int count,
                                int after) {
    // ignored
  }

  @Override
  public void onTextChanged(CharSequence s, int start, int before,
                            int count) {
    // ignored
  }
}
```

We also register the activity itself to be a `TextWatcher`, to find out when the user types something into the `EditText` widget.

`onCreateOptionsMenu()` is where we configure the `ShareActionProvider`, which we obtain by calling `findItem()` on our `Menu` to get the item associated with the provider, then calling `getActionProvider()` on the supplied `MenuItem`. Specifically:

- We supply an `Intent` — configured with the action, MIME type, etc. that we wish to invoke — to `setShareIntent()`
- We supply `MainActivity` itself, as an implementation of `OnShareTargetSelectedListener`, via `setOnShareTargetSelectedListener()`

In the `afterTextChanged()` method needed by the `TextWatcher` interface, we update the `EXTRA_TEXT` extra in the `Intent` to be the current contents of the `EditText`. This way, as the user types, we keep the `Intent` "fresh" with respect to what should be shared. Many consumers of a `ShareActionProvider` will have less dynamic contents, in which case you can just set up the `Intent` up front before you register it with the `ShareActionProvider`.

If the user chooses an item from the `ShareActionProvider`, we are notified via a call to our `onShareTargetSelected()` method. Registering as the `OnShareTargetSelectedListener` is optional — Android will automatically start the selected activity without our involvement. `onShareTargetSelected()` is there if you wish to know the means of sharing that the user chose. In our case, we just flash a `Toast` to indicate that the callback worked.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Broadcasts and Broadcast Receivers

One channel of the `Intent` message bus is used to start activities. A second channel of the `Intent` message bus is used to send broadcasts. As the name suggests, a broadcast `Intent` is one that — by default – is published to any and all applications on the device that wish to tune in.

Sending a Simple Broadcast

The simplest way to send a broadcast `Intent` is to create the `Intent` you want, then call `sendBroadcast()`.

That's it.

At that point, Android will scan through everything set up to tune into a broadcast matching your `Intent`, typically filtering just on the action string. Anyone set up to receive this broadcast will, indeed, receive it, using a `BroadcastReceiver`.

Receiving a Broadcast: In an Activity

To receive such a broadcast in an activity (or a fragment), you will need to do four things.

First, you will need to create an instance of your own subclass of `BroadcastReceiver`. The only method you need to (or should) implement is `onReceive()`, which will be passed the `Intent` that was broadcast, along with a `Context` object that, in this case, you will typically ignore.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Second, you will need to create an instance of an `IntentFilter` object, describing the sorts of broadcasts you want to receive. Most of these filters are set up to watch for a single broadcast `Intent` action, in which case the simple constructor suffices:

```
new IntentFilter(Intent.ACTION_CAMERA_BUTTON)
```

Third, you will need to call `registerReceiver()`, typically from `onResume()` of your activity or fragment, supplying your `BroadcastReceiver` and your `IntentFilter`.

Fourth, you will need to call `unregisterReceiver()`, typically from `onPause()` of your activity or fragment, supplying the same `BroadcastReceiver` instance you provided to `registerReceiver()`.

In between the calls to `registerReceiver()` and `unregisterReceiver()`, you will receive any broadcasts matching the `IntentFilter`.

The biggest downside to this approach is that some activity has to register the receiver. Sometimes, you want to receive broadcasts even when there is no activity around. To do that, you will need to use a different technique: registering the receiver in the manifest.

Receiving a Broadcast: Via the Manifest

You can also tell Android about broadcasts you wish to receive by adding a `<receiver>` element to your manifest, identifying the class that implements your `BroadcastReceiver` (via the `android:name` attribute), plus an `<intent-filter>` that describes the broadcast(s) you wish to receive:

```
<receiver android:name=".OnBootReceiver">
  <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED"/>
  </intent-filter>
</receiver>
```

The good news is that this `BroadcastReceiver` will be available for broadcasts occurring at any time. There is no assumption that you have an activity already running that called `registerReceiver()`.

The bad news is that the instance of the `BroadcastReceiver` used by Android to process a broadcast will live for only so long as it takes to execute the `onReceive()` method. At that point, the `BroadcastReceiver` is discarded. Hence, it is not safe for a manifest-registered `BroadcastReceiver` to do anything that needs to run after

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`onReceive()` itself completes, such as forking a thread. After all, Android may well terminate the process within milliseconds, if there is no other running component in the process.

More bad news: `onReceive()` is called on the main application thread — the same main application thread that handles the UI of all of your activities. And, you are subject to the same limitations as are your activity lifecycle methods and anything else called on the main application thread:

- Any time spent in `onReceive()` will freeze your UI, if you happen to have a foreground activity
- If you spend too long in `onReceive()`, Android will terminate your `BroadcastReceiver` without waiting for `onReceive()` to complete

This makes using a manifest-registered `BroadcastReceiver` a bit tricky. If the work to be done is very quick, just implement it in `onReceive()`. Otherwise, you will probably need to pair this `BroadcastReceiver` with a component known as an `IntentService`, which we will examine in the next chapter.

The Stopped State

On Android 3.1 and higher, when your app is first installed on the device, it is in a "stopped" state. This has nothing to do with `onStop()` of any activity. While in the stopped state, your manifest-registered `BroadcastReceivers` will not receive any broadcasts.

Getting Out of the Stopped State

To get out of the stopped state, something on the app (that itself is not in the stopped state) must use an explicit `Intent` to invoke one of your components.

The most common way this happens is for the user to tap on a launcher icon associated with your launcher activity. Under the covers, the home screen's launcher will create an explicit `Intent`, identifying your activity, and use that with `startActivity()`. This moves you out of the stopped state.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Getting Into the Stopped State

As noted above, you start off in the stopped state. Once you are moved out of the stopped state, via the explicit `Intent`, you will remain out of the stopped state until one of two things happens:

1. The user uninstalls your app
2. The user "force-stops" your app

The latter normally occurs when the user clicks the "Force Stop" button on your app's screen in the Settings app (Settings > Apps). There is some evidence that some device manufacturers have tied their own device's task manager to do a "force stop" when the user removes a task — this was not a particularly wise choice on the part of those manufacturers.

Note that a reboot does *not* move you back into the stopped state. You remain in the normal state through a reboot.

Example System Broadcasts

There are many, many broadcasts sent out by Android itself, which you can tune into if you see fit. Many, but not all, of these are documented on the `Intent` class. The values in the "Constants" table that have "Broadcast Action" leading off their description are action strings used for system broadcasts. There are other such broadcast actions scattered around the SDK, though, so do not assume that they are all documented on `Intent`.

The following sections will examine two of these broadcasts, to see how the `BroadcastReceiver` works in action.

At Boot Time

A popular request is to have code get control when the device is powered on. This is doable but somewhat dangerous, in that too many on-boot requests slow down the device startup and may make things sluggish for the user.

In order to be notified when the device has completed its system boot process, you will need to request the `RECEIVE_BOOT_COMPLETED` permission. Without this, even if you arrange to receive the boot broadcast `Intent`, it will not be dispatched to your receiver.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

As the Android documentation describes it:

> Though holding this permission does not have any security implications, it can have a negative impact on the user experience by increasing the amount of time it takes the system to start and allowing applications to have themselves running without the user being aware of them. As such, you must explicitly declare your use of this facility to make that visible to the user.

We also need to register our `BroadcastReceiver` in the manifest — by the time an activity would call `registerReceiver()`, the boot will have long since occurred.

For example, let us examine the `Intents/OnBoot` sample project.

In our manifest, we request the needed permission and register our `BroadcastReceiver`, along with an activity:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.sysevents.boot"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <supports-screens
    android:largeScreens="false"
    android:normalScreens="true"
    android:smallScreens="false"/>

  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <receiver android:name=".OnBootReceiver">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>
    </receiver>

    <activity
      android:name="BootstrapActivity"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

`OnBootReceiver` simply logs a message to LogCat:

```
package com.commonsware.android.sysevents.boot;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class OnBootReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    Log.d(getClass().getSimpleName(), "Hi, Mom!");
  }
}
```

To test this on Android 3.0 and earlier, simply install the application and reboot the device — you will see the message appear in LogCat.

However, on Android 3.1 and higher, the user must first manually launch some activity before any manifest-registered `BroadcastReceiver` objects will be used, as noted above in the section covering the stopped state. Hence, if you were to just install the application and reboot the device, nothing would happen. The little `BootstrapActivity` is merely there for the user to launch, so that the `ACTION_BOOT_COMPLETED` `BroadcastReceiver` will start working.

On Battery State Changes

One theme with system events is to use them to help make your users happier by reducing your impacts on the device while the device is not in a great state. Most applications are impacted by battery life. Dead batteries run no apps. Hence, knowing the battery level may be important for your app.

There is an `ACTION_BATTERY_CHANGED` `Intent` that gets broadcast as the battery status changes, both in terms of charge (e.g., 80% charged) and charging (e.g., the device is now plugged into AC power). You simply need to register to receive this `Intent` when it is broadcast, then take appropriate steps.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

One of the limitations of ACTION_BATTERY_CHANGED is that you have to use registerReceiver() to set up a BroadcastReceiver to get this Intent when broadcast. You cannot use a manifest-declared receiver. There are separate ACTION_BATTERY_LOW and ACTION_BATTERY_OK broadcasts that you *can* receive from a manifest-registered receiver, but they are broadcast far less frequently, only when the battery level falls below or rises above some undocumented "low" threshold.

To demonstrate ACTION_BATTERY_CHANGED, take a peek at the Intents/OnBattery sample project.

In there, you will find a res/layout/batt.xml resource containing a ProgressBar, a TextView, and an ImageView, to serve as a battery monitor:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <ProgressBar
    android:id="@+id/bar"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <TextView
      android:id="@+id/level"
      android:layout_width="0px"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:textSize="16pt"/>

    <ImageView
      android:id="@+id/status"
      android:layout_width="0px"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
  </LinearLayout>

</LinearLayout>
```

This layout is used by a BatteryFragment, which registers to receive the ACTION_BATTERY_CHANGED Intent in onResume() and unregisters in onPause():

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.battmon;

import android.app.Fragment;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.BatteryManager;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.ProgressBar;
import android.widget.TextView;

public class BatteryFragment extends Fragment {
  private ProgressBar bar=null;
  private ImageView status=null;
  private TextView level=null;

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.batt, parent, false);

    bar=(ProgressBar)result.findViewById(R.id.bar);
    status=(ImageView)result.findViewById(R.id.status);
    level=(TextView)result.findViewById(R.id.level);

    return(result);
  }

  @Override
  public void onResume() {
    super.onResume();

    IntentFilter f=new IntentFilter(Intent.ACTION_BATTERY_CHANGED);

    getActivity().registerReceiver(onBattery, f);
  }

  @Override
  public void onPause() {
    getActivity().unregisterReceiver(onBattery);

    super.onPause();
  }

  BroadcastReceiver onBattery=new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
      int pct=
          100 * intent.getIntExtra(BatteryManager.EXTRA_LEVEL, 1)
              / intent.getIntExtra(BatteryManager.EXTRA_SCALE, 1);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      bar.setProgress(pct);
      level.setText(String.valueOf(pct));

      switch (intent.getIntExtra(BatteryManager.EXTRA_STATUS, -1)) {
        case BatteryManager.BATTERY_STATUS_CHARGING:
          status.setImageResource(R.drawable.charging);
          break;

        case BatteryManager.BATTERY_STATUS_FULL:
          int plugged=
              intent.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);

          if (plugged == BatteryManager.BATTERY_PLUGGED_AC
              || plugged == BatteryManager.BATTERY_PLUGGED_USB) {
            status.setImageResource(R.drawable.full);
          }
          else {
            status.setImageResource(R.drawable.unplugged);
          }
          break;

        default:
          status.setImageResource(R.drawable.unplugged);
          break;
      }
    }
  };
}
```

The key to `ACTION_BATTERY_CHANGED` is in the "extras". Many extras are packaged in the `Intent`, to describe the current state of the battery, such as the following constants defined on the `BatteryManager` class:

- `EXTRA_HEALTH`, which should generally be `BATTERY_HEALTH_GOOD`
- `EXTRA_LEVEL`, which is the proportion of battery life remaining as an integer, specified on the scale described by the `EXTRA_SCALE` value
- `EXTRA_PLUGGED`, which will indicate if the device is plugged into AC power (`BATTERY_PLUGGED_AC`) or USB power (`BATTERY_PLUGGED_USB`)
- `EXTRA_SCALE`, which indicates the maximum possible value of level (e.g., 100, indicating that level is a percentage of charge remaining)
- `EXTRA_STATUS`, which will tell you if the battery is charging (`BATTERY_STATUS_CHARGING`), full (`BATTERY_STATUS_FULL`), or discharging (`BATTERY_STATUS_DISCHARGING`)
- `EXTRA_TECHNOLOGY`, which indicates what sort of battery is installed (e.g., `"Li-Ion"`)
- `EXTRA_TEMPERATURE`, which tells you how warm the battery is, in tenths of a degree Celsius (e.g., 213 is 21.3 degrees Celsius)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `EXTRA_VOLTAGE`, indicating the current voltage being delivered by the battery, in millivolts

In the case of `BatteryFragment`, when we receive an `ACTION_BATTERY_CHANGED` `Intent`, we do three things:

1. We compute the percentage of battery life remaining, by dividing the level by the scale
2. We update the `ProgressBar` and `TextView` to display the battery life as a percentage
3. We display an icon, with the icon selection depending on whether we are charging (status is `BATTERY_STATUS_CHARGING`), full but on the charger (status is `BATTERY_STATUS_FULL` and plugged is `BATTERY_PLUGGED_AC` or `BATTERY_PLUGGED_USB`), or are not plugged in

If you plug this into a device, it will show you the device's charge level:

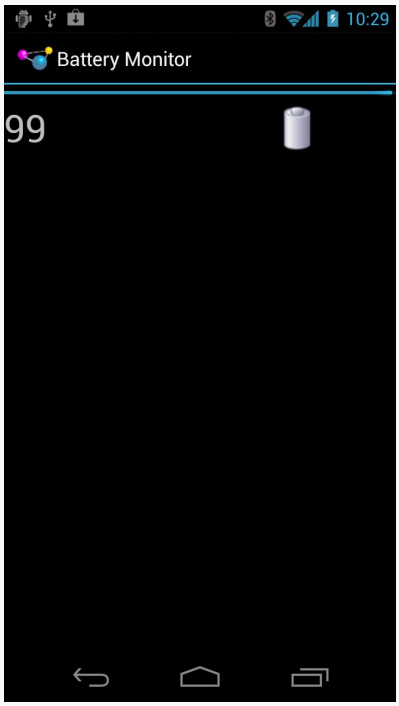


Figure 284: The Battery Monitor

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Sticky Broadcasts and the Battery

NOTE: Sticky broadcasts are deprecated in Android 5.0, and the documentation hints that they may be abandoned entirely in the future.

Android has a notion of "sticky broadcast `Intents`". Normally, a broadcast `Intent` will be delivered to interested parties and then discarded. A sticky broadcast `Intent` is delivered to interested parties and retained until the next matching `Intent` is broadcast. Applications can call `registerReceiver()` with an `IntentFilter` that matches the sticky broadcast, but with a `null` `BroadcastReceiver`, and get the sticky `Intent` back as a result of the `registerReceiver()` call.

This may sound confusing. Let's look at this in the context of the battery.

Earlier in this section, you saw how to register for `ACTION_BATTERY_CHANGED` to get information about the battery delivered to you. You can also, though, get the latest battery information without registering a receiver. Just create an `IntentFilter` to match `ACTION_BATTERY_CHANGED` (as shown above) and call `registerReceiver()` with that filter and a `null` `BroadcastReceiver`. The `Intent` you get back from `registerReceiver()` is the last `ACTION_BATTERY_CHANGED` `Intent` that was broadcast, with the same extras. Hence, you can use this to get the current (or near-current) battery status, rather than having to bother registering an actual `BroadcastReceiver`.

This is why the sample app shows its results immediately — it was given the last-broadcast edition of the `ACTION_BATTERY_CHANGED` broadcast once we called `registerReceiver()`.

Battery and the Emulator

Your emulator does not really have a battery. If you run this sample application on an emulator, you will see, by default, that your device has 50% fake charge remaining and that it is being charged. However, it is charged infinitely slowly, as it will not climb past 50%... at least, not without help.

NOTE: At the time of this writing, the Linux emulator does not properly emulate the battery at all, showing 0% battery charge and not responding to the `telnet` commands described below. In theory, this issue will track progress on resolving this bug.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

While the emulator will only show fixed battery characteristics, you can change what those values are, through the highly advanced user interface known as `telnet`.

You may have noticed that your emulator title bar consists of the name of your AVD plus a number, frequently 5554. That number is not merely some engineer's favorite number. It is also an open port, on your emulator, to which you can `telnet` into, on `localhost` (127.0.0.1) on your development machine.

There are many commands you can issue to the emulator by means of `telnet` . To change the battery level, use `power capacity NN`, where `NN` is the percentage of battery life remaining that you wish the emulator to return. If you do that while you have an `ACTION_BATTERY_CHANGED` `BroadcastReceiver` registered, the receiver will receive a broadcast `Intent`, informing you of the change.

You can also experiment with some of the other power subcommands (e.g., `power ac on` or `power ac off`), or other commands (e.g., `geo`, to send simulated GPS fixes, just as you can do from DDMS).

Battery Data on Android 5.0+

As noted earlier, Android 5.0 deprecates sticky broadcasts. The existing broadcasts still work, though. And, even if someday Android gets rid of sticky broadcasts entirely, broadcasts like `ACTION_BATTERY_CHANGED` most likely will *still* work, albeit just as a regular broadcast.

To get current battery information on Android 5.0 and higher, `BatteryManager` offers `getIntProperty()` and `getLongProperty()`, where the keys for the "properties" are `BATTERY_PROPERTY_*` constants defined on `BatteryManager`, such as `BATTERY_PROPERTY_CAPACITY` to determine the percentage of remaining battery capacity.

The Order of Things

Another variation on the broadcast `Intent` is the ordered broadcast.

Normally, if you broadcast an `Intent`, and there are 10 registered `BroadcastReceivers` that match that `Intent`, all 10 will receive the broadcast, in indeterminate order, and possibly in parallel (particularly on multi-core devices).

With an ordered broadcast, the behavior shifts a bit:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Only one `BroadcastReceiver` at a time will receive the broadcast
- The order in which the `BroadcastReceivers` receive the broadcast is (somewhat) controlled by their developers
- A `BroadcastReceiver` can "abort" the broadcast, preventing other receivers in the chain from receiving it

Sending an ordered broadcast is merely a matter of calling `sendOrderedBroadcast()`.

Receiving an ordered broadcast, at its core, is identical to receiving a regular broadcast: you write a `BroadcastReceiver` and register it via the manifest or `registerReceiver()`. However, you have two additional options when registering that `BroadcastReceiver`.

First, you can specify a priority, either via `setPriority()` on the `IntentFilter` or `android:priority` on the `<intent-filter>` element. The priority is a positive integer, with higher numbers indicating higher priority. Higher-priority receivers will get the broadcast sooner than will lower-priority receivers.

Second, your `BroadcastReceiver` can call `abortBroadcast()` to consume the event, preventing any lower-priority receivers from even seeing the broadcast.

Keeping It Local

A broadcast `Intent`, by default and nearly by definition, is broadcast. Anything on the device could have a receiver "tuned in" to listen for such broadcasts. While you can use `setPackage()` on `Intent` to restrict the distribution, the broadcast still goes through the standard broadcast mechanism, which involves transferring the `Intent` to an OS process, which then does the actual broadcasting. Hence, a broadcast `Intent` has some overhead.

Yet, there are times when using broadcasts within an app is handy, but it would be nice to avoid the overhead. To help with this the core Android team added `LocalBroadcastManager` to the Android Support package, to provide an in-process way of doing broadcasts with the standard `Intent`, `IntentFilter`, and `BroadcastReceiver` classes, yet with less overhead.

`LocalBroadcastManager` is supplied by both the `android-support-v4.jar` and `android-support-v13.jar` libraries. Generally speaking, if your

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`android:minSdkVersion` is less than 13, you probably should choose `android-support-v4.jar`.

The only real difference, from a coding standpoint, in using `LocalBroadcastManager` is that you call `registerReceiver()`, `unregisterReceiver()`, and `sendBroadcast()` on an instance of `LocalBroadcastManager`, instead of on an instance of `Context`. You get the `LocalBroadcastManager` singleton for your process via a static `getInstance()` method on `LocalBroadcastManager` itself.

We will see `LocalBroadcastManager` in use in one of the samples in the services chapter.

Visit the Trails!

We examine `LocalBroadcastManager` in more detail, along with other event bus alternatives, later in the book.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #15 - Sharing Your Notes

Perhaps you would like to get your notes off of our book reader app and into someplace else, or perhaps you would like to share them with somebody else. Either way, we can do that using an `ACTION_SEND` operation, to allow the user to choose how to "send" the notes, such as sending them by email or uploading them to some third-party note service.

To make this work, we will add a `ShareActionProvider` to our action bar on the `NoteFragment`.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Adding a ShareActionProvider

First, we need to allow the user to indicate that they want to "share" the note displayed in the current `NoteFragment`. By putting an action bar item on the activity where the `NoteFragment` is displayed, we do not need to worry about letting the user choose which note to send — we simply send whichever note they happen to be viewing or editing.

By using a `ShareActionProvider`, the action item will handle most of the work for allowing the user to choose where to send the note to. We only need to provide an `Intent` that identifies what is to be shared.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Modify `res/menu/notes.xml` to add in the new `share` toolbar button:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/share"
        android:actionProviderClass="android.widget.ShareActionProvider"
        android:showAsAction="ifRoom"
        android:title="@string/share"/>
    <item
        android:id="@+id/delete"
        android:icon="@drawable/ic_action_discard"
        android:showAsAction="ifRoom|withText"
        android:title="@string/delete">
    </item>
</menu>
```

Note that this menu definition requires a new string resource, named `share`, with a value like `Share`.

Step #2: Sharing the Note

Now, we need to configure the `ShareActionProvider`, in particular supplying it with a continuously-updated `Intent`, based upon what the user has typed into the `EditText`.

Add a `ShareActionProvider` data member to `NoteFragment`, named `share`, along with an `Intent` data member named `shareIntent` configured to use `ACTION_SEND` of a MIME type of `text/plain`:

```
  private ShareActionProvider share=null;
  private Intent shareIntent=
      new Intent(Intent.ACTION_SEND).setType("text/plain");
```

Then, in `onCreateView()`, tell the `EditText` to let us know when the user changes the text, via `addTextChangedListener()`:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.editor, container, false);

    editor=(EditText)result.findViewById(R.id.editor);
    editor.addTextChangedListener(this);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(result);
  }
```

This will fail to compile, as our `NoteFragment` is not implementing the `TextWatcher` interface. So, modify the `NoteFragment` class declaration to include the `TextWatcher` interface:

```
public class NoteFragment extends Fragment implements TextWatcher {
```

That, in turn, will require us to implement three methods:

1. `afterTextChanged()`
2. `beforeTextChanged()`
3. `onTextChanged()`

In our case, we care about `afterTextChanged()`. So, add the following three methods to `NoteFragment`:

```
  @Override
  public void afterTextChanged(Editable s) {
    shareIntent.putExtra(Intent.EXTRA_TEXT, s.toString());
  }

  @Override
  public void beforeTextChanged(CharSequence s, int start, int count,
                                int after) {
    // ignored
  }

  @Override
  public void onTextChanged(CharSequence s, int start, int before,
                            int count) {
    // ignored
  }
```

Here, we update the `shareIntent` with the latest text to be shared, storing it in `EXTRA_TEXT`, per the instructions in the Android developer documentation for working with `ACTION_SEND`.

However, we have not initialized `share` yet. We can do that in `onCreateOptionsMenu()`, adding a call to `findItem()` to find our `R.id.share` menu item, then calling `getActionProvider()` to get the `ShareActionProvider` out of the menu item:

```
  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.notes, menu);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    share=
        (ShareActionProvider)menu.findItem(R.id.share)
                                 .getActionProvider();
    share.setShareIntent(shareIntent);

    super.onCreateOptionsMenu(menu, inflater);
  }
```

Here, we also attach the `shareIntent` to the `ShareActionProvider`, so when it comes time to share the text, the `ShareActionProvider` knows how to do that.

Step #3: Testing the Result

If you run this on a device and navigate to a filled-in note, you will see the new action bar item:

Figure 285: ShareActionProvider in NoteFragment

If you tap on it, you will get a roster of possible ways to share the text:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

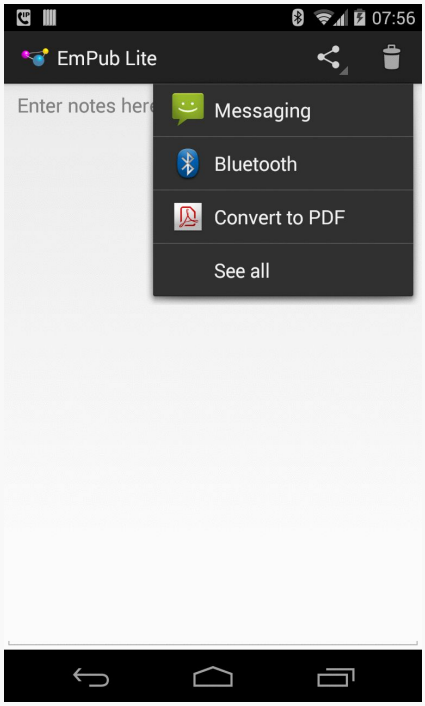


Figure 286: ShareActionProvider in NoteFragment, Expanded

The exact options you see will vary based on your device or emulator, and what apps are installed on it that know how to share plain text.

Unfortunately, your emulator may have nothing that can handle this `Intent`. If that is the case, you will crash with an `ActivityNotFoundException`. To get past this, if you enter `http://goo.gl/w113e` in your emulator's browser, that should allow you to download and install a copy of the APK from the `Intents/FauxSender` sample project that we covered earlier in this book. When the download is complete (which should be very quick), open up the notification drawer and tap on the "download complete" notification. This should begin the installation process. Depending on your Android version, you may also need to "allow installation of non-Market apps" — after fixing this, you can use the Downloads app on the emulator to try installing the APK again. Once FauxSender is installed, it will respond to your attempts to share a note.

In Our Next Episode…

… we will allow the user to update the book's contents over the Internet.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Services and the Command Pattern

As noted previously, Android services are for long-running processes that may need to keep running even when decoupled from any activity. Examples include playing music even if the "player" activity is destroyed, polling the Internet for RSS/Atom feed updates, and maintaining an online chat connection even if the chat client loses focus due to an incoming phone call.

Services are created when manually started (via an API call) or when some activity tries connecting to the service via inter-process communication (IPC). Services will live until specifically shut down or until Android is desperate for RAM and terminates the process. Running for a long time has its costs, though, so services need to be careful not to use too much CPU or keep radios active too much of the time, lest the service cause the device's battery to get used up too quickly.

This chapter outlines the basic theory behind creating and consuming services, including a look at the "command pattern" for services.

Why Services?

Services are a "Swiss Army knife" for a wide range of functions that do not require direct access to an activity's user interface, such as:

1. Performing operations that need to continue even if the user leaves the application's activities, like a long download (as seen with the Play Store) or playing music (as seen with Android music apps)
2. Performing operations that need to exist regardless of activities coming and going, such as maintaining a chat connection in support of a chat application

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

3. Providing a local API to remote APIs, such as might be provided by a Web service
4. Performing periodic work without user intervention, akin to cron jobs or Windows scheduled tasks

Even things like home screen app widgets often involve a service to assist with long-running work.

The primary role of a service is as a flag to the operating system, letting it know that your process is still doing work, despite the fact that it is in the background. This makes it *somewhat* less likely that Android will terminate your process due to low memory conditions.

Many applications will not need any services. Very few applications will need more than one. However, the service is a powerful tool for an Android developer's toolbox and is a subject with which any qualified Android developer should be familiar.

Setting Up a Service

Creating a service implementation shares many characteristics with building an activity. You inherit from an Android-supplied base class, override some lifecycle methods, and hook the service into the system via the manifest.

The Service Class

Just as an activity in your application extends either `Activity` or an Android-supplied `Activity` subclass, a service in your application extends either `Service` or an Android-supplied `Service` subclass. The most common `Service` subclass is `IntentService`, used primarily for the command pattern, described later in this chapter. That being said, many services simply extend `Service`.

Lifecycle Methods

Just as activities have `onCreate()`, `onResume()`, `onPause()` and kin, `Service` implementations have their own lifecycle methods, such as:

- `onCreate()`, which, as with activities, is called when the service process is created, by any means
- `onStartCommand()`, which is called each time the service is sent a command via `startService()`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `onBind()`, which is called whenever a client binds to the service via `bindService()`
- `onDestroy()` which is called as the service is being shut down

As with activities, services initialize whatever they need in `onCreate()` and clean up those items in `onDestroy()`. And, as with activities, the `onDestroy()` method of a service might not be called, if Android terminates the entire application process, such as for emergency RAM reclamation.

The `onStartCommand()` and `onBind()` lifecycle methods will be implemented based on your choice of communicating to the client, as will be explained later in this chapter.

Note that `Service` is an `abstract` class and `onBind()` is an `abstract` method, so even if you are not using `bindService()`, you will need to implement `onBind()` in order to successfully compile. A common approach here is to have `onBind()` simply return `null`.

Manifest Entry

Finally, you need to add the service to your `AndroidManifest.xml` file, for it to be recognized as an available service for use. That is simply a matter of adding a `<service>` element as a child of the `application` element, providing `android:name` to reference your service class.

Since the service class is in the same Java namespace as everything else in this application, we can use the shorthand (e.g., `"PlayerService"`) to reference our class.

For example, here is a manifest showing the `<service>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.fakeplayer"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="14"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Holo.Light.DarkActionBar">
    <activity
      android:name="FakePlayer"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>

    <service android:name="PlayerService"/>
  </application>

</manifest>
```

Communicating To Services

Clients of services — frequently activities, though not necessarily — have two main ways to send requests or information to a service. One approach is to send a command, which creates no lasting connection to the service. The other approach is to bind to the service, establishing a communications channel that lasts as long as the client needs it.

Sending Commands with startService()

The simplest way to work with a service is to call `startService()`. The `startService()` method takes an `Intent` parameter, much like `startActivity()` does. In fact, the `Intent` supplied to `startService()` has the same two-part role as it does with `startActivity()`:

1. Identify the service to communicate with
2. Supply parameters, in the form of `Intent` extras, to tell the service what it is supposed to do

For a local service — the focus of this chapter — the simplest form of `Intent` is one that identifies the class that implements the `Intent` (e.g., `new Intent(this, MyService.class);`).

The call to `startService()` is asynchronous, so the client will not block. The service will be created if it is not already running, and it will receive the `Intent` via a call to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the onStartCommand() lifecycle method. The service can do whatever it needs to in onStartCommand(), but since onStartCommand() is called on the main application thread, it should do its work very quickly. Anything that might take more than a handful of milliseconds should be delegated to a background thread.

The onStartCommand() method can return one of several values, mostly to indicate to Android what should happen if the service's process should be killed while it is running. The most likely return values are:

1. START_STICKY, meaning that the service should be moved back into the started state (as if onStartCommand() had been called), but do not re-deliver the Intent to onStartCommand()
2. START_REDELIVER_INTENT, meaning that the service should be restarted via a call to onStartCommand(), supplying the same Intent as was delivered this time
3. START_NOT_STICKY, meaning that the service should remain stopped until explicitly started by application code

By default, calling startService() not only sends the command, but tells Android to keep the service running until something tells it to stop. One way to stop a service is to call stopService(), supplying the same Intent used with startService(), or at least one that is equivalent (e.g., identifies the same class). At that point, the service will stop and will be destroyed. Note that stopService() does not employ any sort of reference counting, so three calls to startService() will result in a single service running, which will be stopped by a call to stopService().

Another possibility for stopping a service is to have the service call stopSelf() on itself. You might do this if you use startService() to have a service begin running and doing some work on a background thread, then having the service stop itself when that background work is completed.

Binding to Services

Another approach to communicating with a service is to use the binding pattern. Here, instead of packaging commands to be sent via an Intent, you can obtain an actual API from the service, with whatever data types, return values, and so on that you wish. You then invoke that API no different than you would on some local object.

The benefit is the richer API. The cost is that binding is more complex to set up and more complex to maintain, particularly across configuration changes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We will discuss the binding pattern later in this book.

Scenario: The Music Player

Most audio player applications in Android — for music, audiobooks, or whatever — do not require the user to remain in the player application itself. Rather, the user can go on and do other things with their device, with the audio playing in the background.

The sample project reviewed in this section is `Service/FakePlayer`.

The Design

We will use `startService()`, since we want the service to run even when the activity starting it has been destroyed. However, we will use a regular `Service`, rather than an `IntentService`. An `IntentService` is designed to do work and stop itself, whereas in this case, we want the user to be able to stop the music playback when the user wants to.

Since music playback is outside the scope of this chapter, the service will simply stub out those particular operations.

The Service Implementation

Here is the implementation of this `Service`, named `PlayerService`:

```
package com.commonsware.android.fakeplayer;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;

public class PlayerService extends Service {
  public static final String EXTRA_PLAYLIST="EXTRA_PLAYLIST";
  public static final String EXTRA_SHUFFLE="EXTRA_SHUFFLE";
  private boolean isPlaying=false;

  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    String playlist=intent.getStringExtra(EXTRA_PLAYLIST);
    boolean useShuffle=intent.getBooleanExtra(EXTRA_SHUFFLE, false);

    play(playlist, useShuffle);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(START_NOT_STICKY);
  }

  @Override
  public void onDestroy() {
    stop();
  }

  @Override
  public IBinder onBind(Intent intent) {
    return(null);
  }

  private void play(String playlist, boolean useShuffle) {
    if (!isPlaying) {
      Log.w(getClass().getName(), "Got to play()!");
      isPlaying=true;
    }
  }

  private void stop() {
    if (isPlaying) {
      Log.w(getClass().getName(), "Got to stop()!");
      isPlaying=false;
    }
  }
}
```

In this case, we really do not need anything for `onCreate()`, so that lifecycle method is skipped. On the other hand, we have to implement `onBind()`, because that is an abstract method on `Service`.

When the client calls `startService()`, `onStartCommand()` is called in `PlayerService`. Here, we get the `Intent` and pick out some extras to tell us what to play back (`EXTRA_PLAYLIST`) and other configuration details (e.g., `EXTRA_SHUFFLE`). `onStartCommand()` calls `play()`, which simply flags that we are playing and logs a message to LogCat — a real music player would use `MediaPlayer` to start playing the first song in the playlist. `onStartCommand()` returns `START_NOT_STICKY`, indicating that if Android terminates the process (e.g., low memory), it should not restart it once conditions improve.

`onDestroy()` stops the music from playing — theoretically, anyway — by calling a `stop()` method. Once again, this just logs a message to LogCat, plus updates our internal are-we-playing flag.

In the upcoming chapter on notifications, we will revisit this sample and discuss the use of `startForeground()` to make it easier for the user to get back to the music

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

player, plus let Android know that the service is delivering part of the foreground experience and therefore should not be shut down.

Using the Service

The `PlayerFragment` demonstrating the use of `PlayerService` has a very elaborate UI, consisting of two large buttons:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <Button
    android:id="@+id/start"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="@string/start_the_player"/>

  <Button
    android:id="@+id/stop"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="@string/stop_the_player"/>

</LinearLayout>
```

The fragment itself is not much more complex:

```
package com.commonsware.android.fakeplayer;

import android.app.Fragment;
import android.content.Intent;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class PlayerFragment extends Fragment implements
    View.OnClickListener {
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.main, parent, false);

    result.findViewById(R.id.start).setOnClickListener(this);
    result.findViewById(R.id.stop).setOnClickListener(this);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(result);
  }

  @Override
  public void onClick(View v) {
    Intent i=new Intent(getActivity(), PlayerService.class);

    if (v.getId() == R.id.start) {
      i.putExtra(PlayerService.EXTRA_PLAYLIST, "main");
      i.putExtra(PlayerService.EXTRA_SHUFFLE, true);

      getActivity().startService(i);
    }
    else {
      getActivity().stopService(i);
    }
  }
}
```

The `onCreate()` method merely loads the UI. The `onClick()` method constructs an `Intent` with fake values for `EXTRA_PLAYLIST` and `EXTRA_SHUFFLE`, then calls `startService()`. After you press the "Start" button, you will see the corresponding message in LogCat. Similarly, `stopPlayer()` calls `stopService()`, triggering the second LogCat message. Notably, you do not need to keep the activity running in between those button clicks — you can exit the activity via BACK and come back later to stop the service.

Communicating *From* Services

Sending commands to a service, by default, is a one-way street. Frequently, though, we need to get results from our service back to our activity. There are a few approaches for how to accomplish this.

Broadcast Intents

One approach, first mentioned in the chapter on `Intent` filters, is to have the service send a broadcast `Intent` that can be picked up by the activity... assuming the activity is still around and is not paused. The service can call `sendBroadcast()`, supplying an `Intent` that identifies the broadcast, designed to be picked up by a `BroadcastReceiver`. This could be a component-specific broadcast (e.g., `new Intent(this, MyReceiver.class)`), if the `BroadcastReceiver` is registered in the manifest. Or, it can be based on some action string, perhaps one even documented and designed for third-party applications to listen for.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The activity, in turn, can register a `BroadcastReceiver` via `registerReceiver()`, though this approach will only work for `Intent` objects specifying some action, not ones identifying a particular component. But, when the activity's `BroadcastReceiver` receives the broadcast, it can do what it wants to inform the user or otherwise update itself.

However, for local services, this is not a good choice. System broadcasts like this are intrinsically system-wide; for a local service, you should be using a communications channel that is private to your process.

Pending Results

Your activity can call `createPendingResult()`. This returns a `PendingIntent` – an object that represents an `Intent` and the corresponding action to be performed upon that `Intent` (e.g., use it to start an activity). In this case, the `PendingIntent` will cause a result to be delivered to your activity's implementation of `onActivityResult()`, just as if another activity had been called with `startActivityForResult()` and, in turn, called `setResult()` to send back a result.

Since a `PendingIntent` is `Parcelable`, and can therefore be put into an `Intent` extra, your activity can pass this `PendingIntent` to the service. The service, in turn, can call one of several flavors of the `send()` method on the `PendingIntent`, to notify the activity (via `onActivityResult()`) of an event, possibly even supplying data (in the form of an `Intent`) representing that event.

We will be seeing `PendingIntent` used many places later in this book, such as with notifications and `AlarmManager`.

Event Buses

Event bus implementations — like `LocalBroadcastManager` or greenrobot's EventBus — are a great solution for having a service communicate with objects elsewhere within your process. You can have the service raise events (e.g., `NewEmailEvent`, `UploadCompletedEvent`, `MartiansHaveLandedEvent`), which activities or fragments can listen for and respond to.

Messenger

Yet another possibility is to use a `Messenger` object. A `Messenger` sends messages to an activity's `Handler`. Within a single activity, a `Handler` can be used to send

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

messages to itself, as was mentioned briefly in the chapter on threads. However, between components — such as between an activity and a service — you will need a `Messenger` to serve as the bridge.

As with a `PendingIntent`, a `Messenger` is `Parcelable`, and so can be put into an `Intent` extra. The activity calling `startService()` or `bindService()` would attach a `Messenger` as an extra on the `Intent`. The service would obtain that `Messenger` from the `Intent`. When it is time to alert the activity of some event, the service would:

1. Call `Message.obtain()` to get an empty `Message` object
2. Populate that `Message` object as needed, with whatever data the service wishes to pass to the activity
3. Call `send()` on the `Messenger`, supplying the `Message` as a parameter

The `Handler` will then receive the message via `handleMessage()`, on the main application thread, and so can update the UI or whatever is necessary.

Notifications

Another approach is for the service to let the user know directly about the work that was completed. To do that, a service can raise a `Notification` — putting an icon in the status bar and optionally shaking or beeping or something. This technique is covered in an upcoming chapter.

We can also combine these techniques, such as using an event bus event and detecting when nothing in the UI layer receives the event, so we know that we need to display a `Notification`. We will be examining this pattern later in the book as well.

Scenario: The Downloader

If you elect to download something from the Play Store, you are welcome to back out of the Play Store application entirely. This does not cancel the download – the download and installation run to completion, despite no Play Store activity being on-screen.

You may have similar circumstances in your application, from downloading a purchased e-book to downloading a map for a game to downloading a file from some sort of "drop box" file-sharing service. And, perhaps `DownloadManager` is not

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

going to be a great choice, for any number of reasons (e.g., you want to download the file to internal storage).

The sample project reviewed in this section is `Service/Downloader`, which implements such a downloading service.

The Design

This sort of situation is a perfect use for the command pattern and an `IntentService`. The `IntentService` has a background thread, so downloads can take as long as needed. An `IntentService` will automatically shut down when the work is done, so the service will not linger and you do not need to worry about shutting it down yourself. Your activity can simply send a command via `startService()` to the `IntentService` to tell it to go do the work.

Admittedly, things get a bit trickier when you want to have the activity find out when the download is complete. This example will show the use of `LocalBroadcastManager` for this.

Using the Service

The `DownloadFragment` demonstrating the use of `Downloader` has a trivial UI, consisting of one large button:

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/button"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:text="@string/do_the_download"
/>
```

That UI is initialized in `onCreateView()`, as usual:

```
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.main, parent, false);

    b=(Button)result.findViewById(R.id.button);
    b.setOnClickListener(this);

    return(result);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

When the user clicks the button, `onClick()` is called to disable the button (to prevent accidental duplicate downloads) and call `startService()` to send over a command:

```
  @Override
  public void onClick(View v) {
    b.setEnabled(false);

    Intent i=new Intent(getActivity(), Downloader.class);

    i.setData(Uri.parse("http://commonsware.com/Android/excerpt.pdf"));

    getActivity().startService(i);
  }
```

Here, the `Intent` we pass over has the URL of the file to download (in this case, a URL pointing to a PDF).

The Service Implementation

Here is the implementation of this `IntentService`, named `Downloader`:

```
package com.commonsware.android.downloader;

import android.app.IntentService;
import android.content.Intent;
import android.os.Environment;
import android.support.v4.content.LocalBroadcastManager;
import android.util.Log;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class Downloader extends IntentService {
  public static final String ACTION_COMPLETE=
      "com.commonsware.android.downloader.action.COMPLETE";

  public Downloader() {
    super("Downloader");
  }

  @Override
  public void onHandleIntent(Intent i) {
    try {
      File root=
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);

      root.mkdirs();

      File output=new File(root, i.getData().getLastPathSegment());

      if (output.exists()) {
        output.delete();
      }

      URL url=new URL(i.getData().toString());
      HttpURLConnection c=(HttpURLConnection)url.openConnection();

      FileOutputStream fos=new FileOutputStream(output.getPath());
      BufferedOutputStream out=new BufferedOutputStream(fos);

      try {
        InputStream in=c.getInputStream();
        byte[] buffer=new byte[8192];
        int len=0;

        while ((len=in.read(buffer)) >= 0) {
          out.write(buffer, 0, len);
        }

        out.flush();
      }
      finally {
        fos.getFD().sync();
        out.close();
        c.disconnect();
      }

      LocalBroadcastManager.getInstance(this)
                           .sendBroadcast(new Intent(ACTION_COMPLETE));
    }
    catch (IOException e2) {
      Log.e(getClass().getName(), "Exception in download", e2);
    }
  }
}
```

Our business logic is in `onHandleIntent()`, which is called on an Android-supplied background thread, so we can take whatever time we need. Also, when `onHandleIntent()` ends, the `IntentService` will stop itself automatically... assuming no other requests for downloads occurred while `onHandleIntent()` was running. In that case, `onHandleIntent()` is called again for the next download, and so on.

In `onHandleIntent()`, we first set up a `File` object pointing to where we want to download the file. We use `getExternalStoragePublicDirectory()` to find the public folder for downloads. Since this directory may not exist, we need to create it

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

using `mkdirs()`. We then use the `getLastPathSegment()` convenience method on `Uri`, which returns to us the filename portion of a path-style `Uri`. The result is that our `output` `File` object points to a file, named the same as the file we are downloading, in a public folder.

We then go through a typical `HttpUrlConnection` process to connect to the URL supplied via the `Uri` in the `Intent`, streaming the results from the connection (8KB at a time) out to our designated file. Then, we follow the requested recipe to ensure our file is saved:

- `flush()` the stream
- `sync()` the `FileDescriptor` (from `getFD()`)
- `close()` the stream

This recipe was explained back in the chapter on file I/O.

Finally, it would be nice to let somebody know that the download has completed. So, we send a local broadcast `Intent`, with our own custom action (`ACTION_COMPLETE`), using `LocalBroadcastManager`.

Receiving the Broadcast

Our `DownloadFragment` is set up to listen for that local broadcast `Intent`, by registering a local `BroadcastReceiver` in `onResume()` and unregistering it in `onPause()`:

```
@Override
public void onResume() {
  super.onResume();

  IntentFilter f=new IntentFilter(Downloader.ACTION_COMPLETE);

  LocalBroadcastManager.getInstance(getActivity())
                       .registerReceiver(onEvent, f);
}

@Override
public void onPause() {
  LocalBroadcastManager.getInstance(getActivity())
                       .unregisterReceiver(onEvent);

  super.onPause();
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `BroadcastReceiver` itself re-enables our button, plus displays a `Toast` indicating that the download is complete:

```
  private BroadcastReceiver onEvent=new BroadcastReceiver() {
    public void onReceive(Context ctxt, Intent i) {
      b.setEnabled(true);

      Toast.makeText(getActivity(), R.string.download_complete,
                     Toast.LENGTH_LONG).show();
    }
  };
```

Note that if the user leaves the activity (e.g., BACK, HOME), the broadcast will not be received by the activity. There are other ways of addressing this, particularly combining an ordered broadcast with a `Notification`, which we will examine later in this book.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #16 - Updating the Book

The app is designed to ship a copy of the book's chapters as assets, so a user can just download one thing and get everything they need: book and reader.

However, sometimes books get updated. This is a bit less likely with the material being used in this tutorial, as it is rather unlikely that H. G. Wells will rise from the grave to amend *The War of the Worlds*. However, other books, such as Android developer guides written by balding guys, might be updated more frequently.

Most likely, the way you would get those updates is by updating the entire app, so you get improvements to the reader as well. However, another approach would be to be able to download an update to the book as a separate ZIP file. The reader would use the contents of that ZIP file if one has been downloaded, otherwise it will "fall back" to the copy in assets. That is the approach that we will take in this tutorial, to experiment a bit with Internet access and services. Along the way, we will use Retrofit to call a Web service (of sorts) to find out if an update is available.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Adding a Stub DownloadCheckService

There are a few pieces to our download-the-book-update puzzle:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- We need to determine if there is an update available and, if so, where we can find the ZIP file that is the update
- We need to download the update's ZIP file, which could be a fairly large file
- We need to unpack that ZIP file into internal or external storage, so that it is more easily used by the rest of our code and performs more quickly than would dynamically reading the contents out of the ZIP on the fly
- All of that needs to happen in the background from a threading standpoint
- Ideally, all of that could happen either in the foreground or the background from a UI standpoint (i.e., user manually requests an update check, or an update check is performed automatically on a scheduled basis)

To address the first puzzle piece — determining if there is an update available — we can use an `IntentService`. That makes it easy for us to do the work not only in the background from a threading standpoint, but also be able to use it either from the UI or from some sort of background-work scheduler. So, let's add a `DownloadCheckService` to our project.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Service > "Service (IntentService)" from the context menu. Fill in `DownloadCheckService` as the class name and uncheck the "helper methods" checkbox. Click Finish to generate the `DownloadCheckService` class and add an entry for you to the manifest.

Then, replace the generated implementation of `DownloadCheckService` with:

```
package com.commonsware.empublite;

import android.app.IntentService;
import android.content.Intent;

public class DownloadCheckService extends IntentService {
  public DownloadCheckService() {
    super("DownloadCheckService");
  }

  @Override
  protected void onHandleIntent(Intent intent) {
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. Fill in `DownloadCheckService` in the "Name" field. Click the "Browse..." button next to the "Superclass" field and find `IntentService` to set as the superclass. Then, click "Finish" on the new-class dialog to create the `DownloadCheckService` class.

Then, with `DownloadCheckService` open in the editor, paste in the class definition seen in the Android Studio instructions above.

You will also need to add a new `<service>` node to the list of nodes in the Application sub-tab of `AndroidManifest.xml`, pointing to `DownloadCheckService`, following the same approach that we used for activities in this application — just be sure to define a service instead of an activity.

Step #2: Tying the Service Into the Action Bar

To allow the user to manually request that we update the book (if an update is available), we should add a new action bar item to `EmPubLiteActivity`.

First, copy the `01_core_refresh` icons from the `Action Bar Icons/holo_dark/` directory from the "Action Bar Icon Pack" into your project. In the `01_core_refresh` directory, you will see versions of this icon in various densities. Copy those icon files into their corresponding directories in your project. You can do this by dragging the file from your development desktop OS into the project explorer window and dropping it into the proper directory.

Then, modify the `res/menu/options.xml` file to include the following `<item>` element:

```
    <item
        android:id="@+id/update"
        android:icon="@drawable/ic_action_refresh"
        android:showAsAction="ifRoom|withText"
        android:title="@string/download_update">
    </item>
```

Android Studio users can simply edit the XML for this resource. Eclipse users can add this via the structured editor for `res/menu/options.xml`, following the instructions used for other action bar items.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that this menu definition requires a new string resource, named `download_update`, with a value like `Download Update`.

That allows us to add a new `case` to the `switch` statement in `onOptionsItemSelected()` in `EmPubLiteActivity`:

```
      case R.id.update:
        startService(new Intent(this, DownloadCheckService.class));

        return(true);
```

All we do here is send a command to our `DownloadCheckService` to see if a download is available.

Step #3: Defining Our Event

Our `IntentService` will do the work of updating the book in the background. However, we will want to let the rest of the app know when the book is updated. In particular, the `ModelFragment`, if it exists, needs to know that there is a new set of book contents to display. To accomplish this, we can use another event on our `EventBus`, a `BookUpdatedEvent` in this case.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `BookUpdatedEvent` as the name and click OK to create the empty class.

Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. Fill in `BookUpdatedEvent` in the "Name" field. Leave the "Superclass" field alone, as `BookUpdatedEvent` has no explicit superclass. Then, click "Finish" on the new-class dialog to create the empty `BookUpdatedEvent` class.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Step #4: Defining Our JSON

Under the covers, Retrofit uses GSON for parsing the JSON it retrieves from the Web service (or other URL). Hence, just as we needed to define a Java class that models our JSON for the book contents, we need a Java class that models the data we will get from our server as to whether or not a book update is available.

That JSON looks like:

```
{
  "updatedOn": "20120512",
  "updateUrl": "http://misc.commonsware.com/WarOfTheWorlds-Update.zip"
}
```

We can create a `BookUpdateInfo` class that mimics this structure.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `BookUpdateInfo` as the name and click OK to create the empty class.

Then, with `BookUpdateInfo` open in the editor, paste in the following class definition:

```
package com.commonsware.empublite;

public class BookUpdateInfo {
  String updatedOn;
  String updateUrl;
}
```

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. Fill in `BookUpdateInfo` in the "Name" field. Leave the "Superclass" field alone, as `BookUpdateInfo` has no

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

explicit superclass. Then, click "Finish" on the new-class dialog to create the `BookUpdateInfo` class.

Then, with `BookUpdateInfo` open in the editor, paste in the class definition found in the Android Studio instructions above.

Step #5: Defining Our Retrofit Interface

Retrofit then needs a Java *interface* that provides most of the details for how to fetch our JSON and convert it into a Java object. In our case, we will be using an HTTP `GET` operation to retrieve the JSON, and so we will use the Retrofit `@GET` annotation to point to a path on a server pointing to that JSON.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Java Class from the context menu. Fill in `BookUpdateInterface` as the name, switch the "Kind" to be "Interface", and click OK to create the empty interface.

Then, with `BookUpdateInterface` open in the editor, paste in the following interface definition:

```
package com.commonsware.empublite;

import retrofit.http.GET;

public interface BookUpdateInterface {
  @GET("/misc/empublite-update.json")
  BookUpdateInfo update();
}
```

If you prefer, you can view this file's contents in your Web browser via this GitHub link.

Here, we define our interface as having an `update()` method, returning an instance of our `BookUpdateInfo` structure, with the `@GET` annotation pointing to a path where the corresponding JSON can be found on a server to be designated later.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Interface from the context menu. Fill in `BookUpdateInterface` in the "Name" field. Then, click "Finish" on the new-interface dialog to create the `BookUpdateInterface` interface.

Then, with `BookUpdateInterface` open in the editor, paste in the interface definition shown in the Android Studio instructions above.

Step #6: Retrieving Our JSON Via Retrofit

Now, we can actually use Retrofit to retrieve our `BookUpdateInfo` and see if we have a book update.

First, we need to add the `INTERNET` permission to our app, as we are going to be downloading materials from the `INTERNET`.

Android Studio users can add the following `<uses-permission>` element as children of the root `<manifest>` element in `AndroidManifest.xml`:

```
  <uses-permission android:name="android.permission.INTERNET"/>
```

Eclipse users can double-click on `AndroidManifest.xml` and switch over to the Permissions tab. There, click the Add... button and choose to add a new "Uses Permission" entry. In the drop-down that appears on right, choose `android.permission.INTERNET`. Then, save your changes.

Next, in `DownloadCheckService`, add an `OUR_BOOK_DATE` static data member, representing the edit date of the book baked into our APK, in YYYYMMDD format:

```
  private static final String OUR_BOOK_DATE="20120418";
```

Then, add a `getUpdateUrl()` method to `DownloadCheckService`:

```
  private String getUpdateUrl() {
    RestAdapter restAdapter=
        new RestAdapter.Builder().setEndpoint("http://commonsware.com")
                                 .build();
    BookUpdateInterface updateInterface=
        restAdapter.create(BookUpdateInterface.class);
    BookUpdateInfo info=updateInterface.update();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (info.updatedOn.compareTo(OUR_BOOK_DATE) > 0) {
      return(info.updateUrl);
    }

    return(null);
  }
```

Here, we create a Retrofit `RestAdapter`, pointing to the server that is our "Web service" (really a static JSON file, but that does not matter from the standpoint of the client code). We then use the `RestAdapter` to create an instance of a `BookUpdateInterface` implementation, code-generated by Retrofit. We then call `update()` on that object to get our `BookUpdateInfo`. If the date in the `updatedOn` field of our `BookUpdateInfo` is newer than `OUR_BOOK_DATE`, we return the `updateUrl` field of the `BookUpdateInfo`, which will be a URL pointing to a ZIP archive containing the updated book. If the `updatedOn` value is older than `OUR_BOOK_DATE`, we return `null` to signify that no updates are available.

This is not a particularly well-optimized approach. In particular, we never take into account that, once we have downloaded an update, we are only interested in updates newer than the one we downloaded. As it stands, we always compare the `updatedOn` value to `OUR_BOOK_DATE`, not the last `updatedOn` value that we used. A production-grade app would aim to handle this, such as by saving the last-used `updatedOn` value in a `SharedPreferences` and comparing against it, where available.

Finally, update `onHandleIntent()` to call `getUpdateUrl()`:

```
  @Override
  protected void onHandleIntent(Intent intent) {
    try {
      String url=getUpdateUrl();

      if (url != null) {
        // do something really cool here
      }
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(),
            "Exception downloading update", e);
    }
  }
```

Step #7: Downloading the Update

While the above code gets us the URL of the ZIP archive, it does not actually download it. We need more code to accomplish that.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Add a `private static final String` data member named `UPDATE_FILENAME` to `DownloadCheckService`, representing the name of the file to be stored locally representing the downloaded ZIP file:

```
  private static final String UPDATE_FILENAME="book.zip";
```

Then, in `DownloadCheckService`, add the following `download()` method:

```
  private File download(String url) throws MalformedURLException,
                                          IOException {
    File output=new File(getFilesDir(), UPDATE_FILENAME);

    if (output.exists()) {
      output.delete();
    }

    HttpURLConnection c=
        (HttpURLConnection)new URL(url).openConnection();

    FileOutputStream fos=new FileOutputStream(output.getPath());
    BufferedOutputStream out=new BufferedOutputStream(fos);

    try {
      InputStream in=c.getInputStream();
      byte[] buffer=new byte[16384];
      int len=0;

      while ((len=in.read(buffer)) > 0) {
        out.write(buffer, 0, len);
      }

      out.flush();
    }
    finally {
      fos.getFD().sync();
      out.close();
      c.disconnect();
    }

    return(output);
  }
```

This is your typical sort of `HttpURLConnection` download logic, downloading the contents of the designated URL into a `File` in our internal storage (`getFilesDir()`), with the `UPDATE_FILENAME` as the file's name.

Then, update `onHandleIntent()` in `DownloadCheckService` to call `download()` when we have something to download:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@Override
protected void onHandleIntent(Intent intent) {
  try {
    String url=getUpdateUrl();

    if (url != null) {
      File book=download(url);

      // do something almost as cool here

      book.delete();
    }
  }
  catch (Exception e) {
    Log.e(getClass().getSimpleName(),
          "Exception downloading update", e);
  }
}
```

Here, we delete the file after downloading it, so we do not clutter up our internal storage with the downloaded ZIP. This would appear to defeat the purpose of downloading the ZIP file in the first place, but we will add some code to use the ZIP file in the next step of the tutorial.

Step #8: Unpacking the Update

The last step in the book-download process is to unpack the ZIP archive onto internal storage, so we can start using the downloaded contents.

Add the following `unzip()` method to `DownloadCheckService`:

```
private static void unzip(File src, File dest) throws IOException {
  InputStream is=new FileInputStream(src);
  ZipInputStream zis=new ZipInputStream(new BufferedInputStream(is));
  ZipEntry ze;

  dest.mkdirs();

  while ((ze=zis.getNextEntry()) != null) {
    byte[] buffer=new byte[16384];
    int count;
    FileOutputStream fos=
        new FileOutputStream(new File(dest, ze.getName()));
    BufferedOutputStream out=new BufferedOutputStream(fos);

    try {
      while ((count=zis.read(buffer)) != -1) {
        out.write(buffer, 0, count);
      }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      out.flush();
    }
    finally {
      fos.getFD().sync();
      out.close();
    }

    zis.closeEntry();
  }

  zis.close();
}
```

This is a fairly typical Java recipe for unZIPping a ZIP file (`src`) into a destination directory (`dest`).

Next, add a `public static final String` data member named `UPDATE_BASEDIR` to `DownloadCheckService`:

```
public static final String UPDATE_BASEDIR="updates";
```

This will point to the directory on internal storage where the latest book update will reside.

Then, update `onHandleIntent()` on `DownloadCheckService` once again, this time to add in the call to `unzip()` and some other necessary changes:

```
@Override
protected void onHandleIntent(Intent intent) {
  try {
    String url=getUpdateUrl();

    if (url != null) {
      File book=download(url);
      File updateDir=new File(getFilesDir(), UPDATE_BASEDIR);

      updateDir.mkdirs();
      unzip(book, updateDir);
      book.delete();
      EventBus.getDefault().post(new BookUpdatedEvent());
    }
  }
  catch (Exception e) {
    Log.e(getClass().getSimpleName(), "Exception downloading update",
          e);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here, we:

- Create the `UPDATE_BASEDIR` directory if it does not already exist
- Call `unzip()` to unZIP the ZIP file into that directory
- Post a `BookUpdatedEvent` to signify that a book update is ready

This is a bit sloppy. It is possible that different book updates will have different files, and our `UPDATE_BASEDIR` will have some extra files as a result. Ideally, we should clean out `UPDATE_BASEDIR` before unpacking the ZIP archive. Adding in some recursive delete-all-the-files-in-a-directory logic is left as an exercise for the reader.

At this point, `DownloadCheckService` should resemble:

```
package com.commonsware.empublite;

import android.app.IntentService;
import android.content.Intent;
import android.util.Log;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;
import de.greenrobot.event.EventBus;
import retrofit.RestAdapter;

public class DownloadCheckService extends IntentService {
  private static final String OUR_BOOK_DATE="20120418";
  private static final String UPDATE_FILENAME="book.zip";
  public static final String UPDATE_BASEDIR="updates";

  public DownloadCheckService() {
    super("DownloadCheckService");
  }

  @Override
  protected void onHandleIntent(Intent intent) {
    try {
      String url=getUpdateUrl();

      if (url != null) {
        File book=download(url);
        File updateDir=new File(getFilesDir(), UPDATE_BASEDIR);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      updateDir.mkdirs();
      unzip(book, updateDir);
      book.delete();
      EventBus.getDefault().post(new BookUpdatedEvent());
    }
  }
  catch (Exception e) {
    Log.e(getClass().getSimpleName(), "Exception downloading update",
          e);
  }
}

private String getUpdateUrl() {
  RestAdapter restAdapter=
      new RestAdapter.Builder().setEndpoint("http://commonsware.com")
                               .build();
  BookUpdateInterface updateInterface=
      restAdapter.create(BookUpdateInterface.class);
  BookUpdateInfo info=updateInterface.update();

  if (info.updatedOn.compareTo(OUR_BOOK_DATE) > 0) {
    return(info.updateUrl);
  }

  return(null);
}

private File download(String url) throws MalformedURLException,
                                         IOException {
  File output=new File(getFilesDir(), UPDATE_FILENAME);

  if (output.exists()) {
    output.delete();
  }

  HttpURLConnection c=
      (HttpURLConnection)new URL(url).openConnection();

  FileOutputStream fos=new FileOutputStream(output.getPath());
  BufferedOutputStream out=new BufferedOutputStream(fos);

  try {
    InputStream in=c.getInputStream();
    byte[] buffer=new byte[16384];
    int len=0;

    while ((len=in.read(buffer)) > 0) {
      out.write(buffer, 0, len);
    }

    out.flush();
  }
  finally {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      fos.getFD().sync();
      out.close();
      c.disconnect();
    }

    return(output);
  }

  private static void unzip(File src, File dest) throws IOException {
    InputStream is=new FileInputStream(src);
    ZipInputStream zis=new ZipInputStream(new BufferedInputStream(is));
    ZipEntry ze;

    dest.mkdirs();

    while ((ze=zis.getNextEntry()) != null) {
      byte[] buffer=new byte[16384];
      int count;
      FileOutputStream fos=
          new FileOutputStream(new File(dest, ze.getName()));
      BufferedOutputStream out=new BufferedOutputStream(fos);

      try {
        while ((count=zis.read(buffer)) != -1) {
          out.write(buffer, 0, count);
        }

        out.flush();
      }
      finally {
        fos.getFD().sync();
        out.close();
      }

      zis.closeEntry();
    }

    zis.close();
  }
}
```

Step #9: Using the Update

All this work is nice. However, nothing else in the app knows about this `UPDATE_BASEDIR` copy of the book to actually display it.

In fact, we have two scenarios to consider:

- The user taps the update action bar item, and we download the update and want to show the updated book to the user right now

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Later on, when the user opens the book, we need to realize that we already have an update and use it, rather than using the copy baked into the APK

That will require some changes to our data model, how we populate it from `ModelFragment`, and how we use the results in our `ContentsAdapter`.

First, add a `File baseDir` data member to `BookContents`, along with an accompanying setter method:

```
File baseDir=null;

void setBaseDir(File baseDir) {
  this.baseDir=baseDir;
}
```

Then, add a `getChapterPath()` method to `BookContents` that uses `getChapterFile()` for getting the relative path from the book's JSON, then uses that in conjunction with `baseDir` or the `android_asset` path to come up with a full `WebView`-friendly path to the file, whether it is in assets or a local file:

```
String getChapterPath(int position) {
  String file=getChapterFile(position);

  if (baseDir == null) {
    return("file:///android_asset/book/" + file);
  }

  return(Uri.fromFile(new File(baseDir, file)).toString());
}
```

Next, change the `getItem()` method on `ContentsAdapter` to use this new `getChapterPath()` method on `BookContents`:

```
@Override
public Fragment getItem(int position) {
  return(SimpleContentFragment.newInstance(contents.getChapterPath(position)));
}
```

Then, modify the `run()` method of the `LoadThread` in `ModelFragment` to try to use the update:

```
  @Override
  public void run() {
    prefs=PreferenceManager.getDefaultSharedPreferences(ctxt);

    Gson gson=new Gson();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    File baseDir=
        new File(ctxt.getFilesDir(),
                 DownloadCheckService.UPDATE_BASEDIR);

    try {
      InputStream is;

      if (baseDir.exists()) {
        is=new FileInputStream(new File(baseDir, "contents.json"));
      }
      else {
        is=ctxt.getAssets().open("book/contents.json");
      }

      BufferedReader reader=
          new BufferedReader(new InputStreamReader(is));

      contents=gson.fromJson(reader, BookContents.class);
      is.close();

      if (baseDir.exists()) {
        contents.setBaseDir(baseDir);
      }

      EventBus.getDefault().post(new BookLoadedEvent(contents));
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
    }
  }
```

Here, we do the following, in addition to our original logic:

- See if the `UPDATE_BASEDIR` directory exists or not
- If it does, we use the `contents.json` in it; otherwise, we fall back to the one in `assets/` as before
- Update the `BookContents` with the update directory if we used that for loading the contents

This will handle the case where an update exists when we fire up the app and go to view the book. However, we still need some code that responds to the `BookUpdatedEvent` and arranges to use the updated contents at that point.

With that in mind, augment `onAttach()` on `ModelFragment` to register with the `EventBus`:

```
  @Override
  public void onAttach(Activity host) {
    super.onAttach(host);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    EventBus.getDefault().register(this);

    if (contents == null) {
      new LoadThread(host).start();
    }
  }
```

We also now need a corresponding `onDetach()` method on `ModelFragment` to unregister from the `EventBus`:

```
  @Override
  public void onDetach() {
    EventBus.getDefault().unregister(this);

    super.onDetach();
  }
```

Finally, we can respond to the `BookUpdatedEvent`, via a new `onEventBackgroundThread()` method on `ModelFragment`:

```
  public void onEventBackgroundThread(BookUpdatedEvent event) {
    if (getActivity() != null) {
      new LoadThread(getActivity()).start();
    }
  }
```

The name `onEventBackgroundThread()` signals to the `EventBus` that we want to receive this event on a background thread. In our case, the event is posted on a background thread (the one from the `IntentService`). Hence, our `onEventBackgroundThread()` method is called on that thread. If, however, we were to post a `BookUpdatedEvent` from the main application thread, `EventBus` would deliver our `BookUpdatedEvent` to `onEventBackgroundThread()` on an `EventBus`-supplied background thread, to ensure that we do not tie up the main application thread.

Here, we just kick off a fresh `LoadThread` to reload the `BookContents`, assuming that the user has not just pressed BACK or otherwise destroyed our activity. The new `LoadThread` will see that the update is available and use it, posting its own event to have our UI layer apply the update to the screen.

At this point, if you build and run the app, you will see the update action bar item:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 287: The New Action Bar Item

Pressing that and waiting a moment should cause your book to be updated with new contents downloaded from the Internet:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 288: The Updated Content

In Our Next Episode…

… we will update the book's contents ourselves, periodically in the background.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

AlarmManager and the Scheduled Service Pattern

Many applications have the need to get control every so often to do a bit of work. And, many times, those applications need to get control in the background, regardless of what the user may be doing (or not doing) at the time.

The solution, in most cases, is to use `AlarmManager`, which is roughly akin to `cron` on Linux and OS X and Scheduled Tasks in Windows. You teach `AlarmManager` when you want to get control back, and `AlarmManager` will give you control at that time.

Scenarios

The two main axes to consider with scheduled work are frequency and foreground (vs. background).

If you have an activity that needs to get control every second, the simplest approach is to use a `postDelayed()` loop, scheduling a `Runnable` to be invoked after a certain delay, where the `Runnable` reschedules itself to be invoked after the delay in addition to doing some work. We saw this in the chapter on threads. This has the advantages of giving you control back on the main application thread and avoiding the need for any background threads.

On the far other end of the spectrum, you may need to get control on a somewhat slower frequency (e.g., every 15 minutes), and do so in the background, even if nothing of your app is presently running. You might need to poll some Web server for new information, such as downloading updates to an RSS feed. This is the scenario that `AlarmManager` excels at. While `postDelayed()` works *inside* your process (and therefore does not work if you no longer have a process), `AlarmManager`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

maintains its schedule *outside* of your process. Hence, it can arrange to give you control, even if it has to start up a new process for you along the way.

Options

There are a variety of things you will be able to configure about your scheduled alarms with `AlarmManager`.

Wake Up… Or Not?

The biggest one is whether or not the scheduled event should wake up the device.

A device goes into a sleep mode shortly after the screen goes dark. During this time, nothing at the application layer will run, until something wakes up the device. Waking up the device does not necessarily turn on the screen — it may just be that the CPU starts running your process again.

If you choose a "wakeup"-style alarm, Android will wake up the device to give you control. This would be appropriate if you need this work to occur even if the user is not actively using the device, such as your app checking for critical email messages in the middle of the night. However, it does drain the battery some.

Alternatively, you can choose an alarm that will not wake up the device. If your desired time arrives and the device is asleep, you will not get control until something else wakes up the device.

Repeating… Or Not?

You can create a "one-shot" alarm, to get control once at a particular time in the future. Or, you can create an alarm that will give you control periodically, at a fixed period of your choice (e.g., every 15 minutes).

If you need to get control at multiple times, but the schedule is irregular, use a "one-shot" alarm for the nearest time, where you do your work *and* schedule a "one-shot" alarm for the next-nearest time. This would be appropriate for scenarios like a calendar application, where you need to let the user know about upcoming appointments, but the times for those appointments may not have any fixed schedule.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, for most polling operations (e.g., checking for new messages every NN minutes), a repeating alarm will typically be the better answer.

Inexact… Or Not?

If you do choose a repeating alarm, you will have your choice over having (relatively) precise control over the timing of event or not.

If you choose an "inexact" alarm, while you will provide Android with a suggested time for the first event and a period for subsequent events, Android reserves the right to shift your schedule somewhat, so it can process your events and others around the same time. This is particularly important for "wakeup"-style alarms, as it is more power-efficient to wake up the device fewer times, so Android will try to combine multiple apps' events to be around the same time to minimize the frequency of waking up the device.

However, inexact alarms are annoying to test and debug, simply because you do not have control over when they will be invoked. Hence, during development, you might start with an exact alarm, then switch to inexact alarms once most of your business logic is debugged.

Note that Android 4.4 changes the behavior of `AlarmManager`, such that it is more difficult to actually create an exact-repeating alarm schedule. This will be examined in greater detail shortly, as we review the various methods and flags for scheduling `AlarmManager` events.

Absolute Time… Or Not?

As part of the alarm configuration, you will tell Android when the event is to occur (for one-shot alarms) or when the event is to *first* occur (for repeating alarms). You can provide that time in one of two ways:

- An absolute "real-time clock" time (e.g., 4am tomorrow), or
- A time relative to now

For most polling operations, particularly for periods more frequent than once per day, specifying the time relative to now is easiest. However, some alarms may need to tie into "real world time", such as alarm clocks and calendar alerts — for those, you will need to use the real-time clock (typically by means of a Java `Calendar` object) to indicate when the event should occur.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What Happens (Or Not???)

And, of course, you will need to tell Android what to do when each of these timer events occurs. You will do that in the form of supplying a `PendingIntent`. First mentioned in the chapter on services, a `PendingIntent` is a `Parcelable` object, one that indicates an operation to be performed upon an `Intent`:

- start an activity
- start a service
- send a broadcast

While the service chapter discussed an Android activity using `createPendingResult()` to craft such a `PendingIntent`, that is usually not very useful for `AlarmManager`, as the `PendingIntent` will only be valid so long as the activity is in the foreground. Instead, there are static factory methods on `PendingIntent` that you will use instead (e.g., `getBroadcast()` to create a `PendingIntent` that calls `sendBroadcast()` on a supplied `Intent`). That being said, our next sample will use `createPendingResult()`, to keep the sample as simple as possible.

A Simple Example

A trivial sample app using `AlarmManager` can be found in `AlarmManager/Simple`.

This application consists of a single activity, `SimpleAlarmDemoActivity`, that will both set up an alarm schedule and respond to alarms:

```
package com.commonsware.android.alarm;

import android.app.Activity;
import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.os.SystemClock;
import android.widget.Toast;

public class SimpleAlarmDemoActivity extends Activity {
  private static final int ALARM_ID=1337;
  private static final int PERIOD=5000;
  private PendingIntent pi=null;
  private AlarmManager mgr=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    mgr=(AlarmManager)getSystemService(ALARM_SERVICE);
    pi=createPendingResult(ALARM_ID, new Intent(), 0);
    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME,
                     SystemClock.elapsedRealtime() + PERIOD, PERIOD, pi);
  }

  @Override
  public void onDestroy() {
    mgr.cancel(pi);

    super.onDestroy();
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode == ALARM_ID) {
      Toast.makeText(this, R.string.toast, Toast.LENGTH_SHORT).show();
    }
  }
}
```

In `onCreate()`, in addition to setting up the "hello, world"-ish UI, we:

- Obtain an instance of `AlarmManager`, by calling `getSystemService()`, asking for the `ALARM_SERVICE`, and casting the result to be an `AlarmManager`
- Create a `PendingIntent` by calling `createPendingResult()`, supplying an empty `Intent` as our "result" (since we do not really need it here)
- Calling `setRepeating()` on `AlarmManager`

The call to `setRepeating()` is a bit complex, taking four parameters:

1. The type of alarm we want, in this case `ELAPSED_REALTIME`, indicating that we want to use a relative time base for when the first event should occur (i.e., relative to now) and that we do not need to wake up the device out of any sleep mode
2. The time when we want the first event to occur, in this case specified as a time delta in milliseconds (`PERIOD`) added to "now" as determined by `SystemClock.elapsedRealtime()` (the number of milliseconds since the device was last rebooted)
3. The number of milliseconds to occur between events
4. The `PendingIntent` to invoke for each of these events

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

When the event occurs, since we used `createPendingResult()` to create the `PendingIntent`, our activity gets control in `onActivityResult()`, where we simply display a `Toast` (if the event is for our alarm's request ID). This continues until the activity is destroyed (e.g., pressing the BACK button), at which time we `cancel()` the alarm, supplying a `PendingIntent` to indicate which alarm to cancel. While here we use the same `PendingIntent` object as we used for scheduling the alarm, that is not required — it merely has to be an *equivalent* `PendingIntent`, meaning:

- The `Intent` inside the `PendingIntent` matches the scheduled alarm's `Intent`, in terms of component, action, data (`Uri`), MIME type, and categories
- The ID of the `PendingIntent` (here, `ALARM_ID`) must also match

Running this simply brings up a `Toast` every five seconds until you BACK out of the activity.

The Five set…() Varieties

There are five methods that you can call on `AlarmManager` to establish an alarm, including the `setRepeating()` demonstrated above.

On Android 4.4 (API Level 19) and higher, `setExact()` is used for a one-shot alarm, where you want to get control at one specific time in the future. This would be used for specific events or for irregular alarm schedules.

On Android 4.3 and below, and for apps whose `android:targetSdkVersion` is set to 18 or lower, `set()` has the same behavior as `setExact()`. However, on Android 4.4 and above, apps with their `android:targetSdkVersion` set to be 19 or higher will have different, *inexact* behavior for `set()`. The time of the event is considered a minimum — your `PendingIntent` will not be invoked before your desired time, but it can occur any time thereafter... and you do not have control over how long that delay will be. As with all "inexact" schedules, the objective is for Android to be able to "batch" these events, to do several around the same time, for greater efficiency, particularly when waking up the device.

On Android 4.4 and higher, you have a `setWindow()` option that is a bit of a hybrid between the new-style `set()` and `setExact()`. Here, you specify the time you want the event to occur and an amount of time that Android can "flex" the actual event. So, for example, you might set up an event to occur every hour, with a "window" of five minutes, to allow Android the flexibility to invoke your `PendingIntent` within that five-minute window. This allows for better battery optimization than with

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`setExact()`, while still giving you some control over how far "off the mark" the event can occur.

On Android 4.3 and below, and for apps whose `android:targetSdkVersion` is set to 18 or lower, `setRepeating()` is used for an alarm that should occur at specific points in time at a specific frequency. In addition to specifying the time of the first event, you also specify the period for future events. Android will endeavor to give you control at precisely those times, though since Android is not a real-time operating system (RTOS), microsecond-level accuracy is certainly not guaranteed.

`setInexactRepeating()` is used for an alarm that should occur on a general frequency, such as every 15 minutes. In addition to specifying the time of the first event, you also specify a general frequency, as one of the following public static data members on `AlarmManager`:

- `INTERVAL_FIFTEEN_MINUTES`
- `INTERVAL_HALF_HOUR`
- `INTERVAL_HOUR`
- `INTERVAL_HALF_DAY`
- `INTERVAL_DAY`

Android guarantees that it will give your app control somewhere during that time window, but precisely *when* within that window is up to Android.

Note that on Android 4.4 and above, for apps with their `android:targetSdkVersion` set to be 19 or higher, `setRepeating()` behaves identically to `setInexactRepeating()` – in other words, all repeating alarms are inexact. The only way to get exact repeating would be to use `setExact()` and to re-schedule the event yourself, rather than relying upon Android doing that for you automatically. Ideally, you use `setInexactRepeating()`, to help extend battery life.

The Four Types of Alarms

In the above sample, we used `ELAPSED_REALTIME` as the type of alarm. There are three others:

- `ELAPSED_REALTIME_WAKEUP`
- `RTC`
- `RTC_WAKEUP`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Those with `_WAKEUP` at the end will wake up a device out of sleep mode to execute the `PendingIntent` — otherwise, the alarm will wait until the device is awake for other means.

Those that begin with `ELAPSED_REALTIME` expect the second parameter to `setRepeating()` to be a timestamp based upon `SystemClock.elapsedRealtime()`. Those that begin with `RTC`, however, expect the second parameter to be based upon `System.currentTimeMillis()`, the classic Java "what is the current time in milliseconds since the Unix epoch" method.

When to Schedule Alarms

The sample, though, begs a bit of a question: when are we supposed to set up these alarms? The sample just does so in `onCreate()`, but is that sufficient?

For most apps, the answer is "no". Here are the three times that you will need to ensure that your alarms get scheduled:

When User First Runs Your App

When your app is first installed, none of your alarms are set up, because your code has not yet run to schedule them. There is no means of setting up alarm information in the manifest or something that might automatically kick in.

Hence, you will need to schedule your alarms when the user first runs your app.

As a simplifying measure — and to cover another scenario outlined below — you might be able to simply get away with scheduling your alarms *every* time the user runs your app, as the sample app shown above does. This works for one-shot alarms (using `set()`) and for alarms with short polling periods, and it works because setting up a new alarm schedule for an equivalent `PendingIntent` will replace the old schedule. However, for repeating alarms with slower polling periods, it may excessively delay your events. For example, suppose you have an alarm set to go off every 24 hours, and the user happens to run your app 5 minutes before the next event was to occur — if you blindly reschedule the alarm, instead of going off in 5 minutes, it might not go off for another 24 hours.

There are more sophisticated approaches for this (e.g., using a `SharedPreferences` value to determine if your app has run before or not).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

On Boot

The alarm schedule for alarm manager is wiped clean on a reboot, unlike `cron` or Windows Scheduled Tasks. Hence, you will need to get control at boot time to re-establish your alarms, if you want them to start up again after a reboot. We saw how to get control at boot time, via an `ACTION_BOOT_COMPLETED` `BroadcastReceiver`, back in the chapter on broadcasts.

After a Force-Stop

There are other events that could cause your alarms to become unscheduled. The best example of this is if the user goes into the Settings app and presses "Force Stop" for your app. At this point, on Android 3.1+, nothing of your code will run again, until the user manually launches some activity of yours.

If you are rescheduling your alarms every time your app runs, this will be corrected the next time the user launches your app. And, by definition, you cannot do anything until the user runs one of your activities, anyway.

If you are trying to avoid rescheduling your alarms on each run, though, you have a couple of options.

One is to record the time when your alarm-triggered events occur, each time they occur, such as by updating a `SharedPreference`. When the user launches one of your activities, you check the last-event time — if it was too long ago (e.g., well over your polling period), you assume that the alarm had been canceled, and you reschedule it.

Another is to rely on `FLAG_NO_CREATE`. You can pass this as a parameter to any of the `PendingIntent` factory methods, to indicate that Android should only return an *existing* `PendingIntent` if there is one, and not create one if there is not:

```
PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i,
PendingIntent.FLAG_NO_CREATE);
```

If the `PendingIntent` is `null`, your alarm has been canceled — otherwise, Android would already have such a `PendingIntent` and would have returned it to you. This feels a bit like a side-effect, so we cannot rule out the possibility that, in future versions of Android, this technique could result in false positives (`null` `PendingIntent` despite the scheduled alarm) or false negatives (non-`null` `PendingIntent` despite a canceled alarm).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Archetype: Scheduled Service Polling

The classic `AlarmManager` scenario is where you want to do a chunk of work, in the background, on a periodic basis. This is fairly simple to set up in Android, though perhaps not quite as simple as you might think.

The Main Application Thread Strikes Back

When an `AlarmManager`-triggered event occurs, it is very likely that your application is not running. This means that the `PendingIntent` is going to have to start up your process to have you do some work. Since everything that a `PendingIntent` can do intrinsically gives you control on your main application thread, you are going to have to determine how you want to move your work to a background thread.

One approach is to use a `PendingIntent` created by `getService()`, and have it send a command to an `IntentService` that you write. Since `IntentService` does its work on a background thread, you can take whatever time you need, without interfering with the behavior of the main application thread. This is particularly important when:

- The `AlarmManager`-triggered event happens to occur when the user happens to have one of your activities in the foreground, so you do not freeze the UI, or
- You want the same business logic to be executed on demand by the user, such as via an action bar item, as once again you do not want to freeze the UI

Examining a Sample

An incrementally-less-trivial sample app using `AlarmManager` for the scheduled service pattern can be found in `AlarmManager/Scheduled`.

This application consists of three components: a `BroadcastReceiver`, a `Service`, and an `Activity`.

This sample demonstrates scheduling your alarms at two points in your app:

- At boot time
- When the user runs the activity

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For the boot-time scenario, we need a `BroadcastReceiver` set up to receive the `ACTION_BOOT_COMPLETED` broadcast, with the appropriate permission. So, we set that up, along with our other components, in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.schedsvc"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name=".ScheduledServiceDemoActivity"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>

    <receiver android:name="PollReceiver">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>
    </receiver>

    <service android:name="ScheduledService">
    </service>
  </application>

</manifest>
```

The `PollReceiver` has its `onReceive()` method, to be called at boot time, which delegates its work to a `scheduleAlarms()` static method, so that logic can also be used by our activity:

```
package com.commonsware.android.schedsvc;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;

public class PollReceiver extends BroadcastReceiver {
  private static final int PERIOD=5000;

  @Override
  public void onReceive(Context ctxt, Intent i) {
    scheduleAlarms(ctxt);
  }

  static void scheduleAlarms(Context ctxt) {
    AlarmManager mgr=
        (AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE);
    Intent i=new Intent(ctxt, ScheduledService.class);
    PendingIntent pi=PendingIntent.getService(ctxt, 0, i, 0);

    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME,
                     SystemClock.elapsedRealtime() + PERIOD, PERIOD, pi);
  }
}
```

The `scheduleAlarms()` method retrieves our `AlarmManager`, creates a `PendingIntent` designed to call `startService()` on our `ScheduledService`, and schedules an exact repeating alarm to have that command be sent every five seconds.

The `ScheduledService` itself is the epitome of "trivial", simply logging a message to LogCat on each command:

```
package com.commonsware.android.schedsvc;

import android.app.IntentService;
import android.content.Intent;
import android.util.Log;

public class ScheduledService extends IntentService {
  public ScheduledService() {
    super("ScheduledService");
  }

  @Override
  protected void onHandleIntent(Intent intent) {
    Log.d(getClass().getSimpleName(), "I ran!");
  }
}
```

That being said, because this is an `IntentService`, we could do much more in `onHandleIntent()` and not worry about tying up the main application thread.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Our activity — `ScheduledServiceDemoActivity` — is set up with `Theme.NoDisplay` in the manifest, never calls `setContentView()`, and calls `finish()` right from `onCreate()`. As a result, it has no UI. It simply calls `scheduleAlarms()` and raises a `Toast` to indicate that the alarms are indeed scheduled:

```
package com.commonsware.android.schedsvc;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Toast;

public class ScheduledServiceDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    PollReceiver.scheduleAlarms(this);

    Toast.makeText(this, R.string.alarms_scheduled, Toast.LENGTH_LONG)
         .show();
    finish();
  }
}
```

On Android 3.1+, we also need this activity to move our application out of the stopped state and allow that boot-time `BroadcastReceiver` to work.

If you run this app on a device or emulator, after seeing the `Toast`, messages will appear in LogCat every five seconds, even though you have no activity running.

Staying Awake at Work

The sample shown above works... most of the time.

However, it has a flaw: the device might fall asleep before our service can complete its work, if we woke it up out of sleep mode to process the event.

To understand where this flaw would appear, and to learn how to address it, we need to think a bit more about the event flows and timing of the code we are executing.

Mind the Gap

For a `_WAKEUP`-style alarm, Android makes precisely one guarantee: *if* the `PendingIntent` supplied to `AlarmManager` for the alarm is one created by `getBroadcast()` to send a broadcast `Intent`, Android will ensure that the device will

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

stay awake long enough for `onReceive()` to be completed. Anything beyond that is not guaranteed.

In the sample shown above, we are not using `getBroadcast()`. We are taking the more straightforward approach of sending the command directly to the service via a `getService()` `PendingIntent`. Hence, Android makes no guarantees about what happens after `AlarmManager` wakes up the device, and the device could fall back asleep before our `IntentService` completes processing of `onHandleIntent()`.

The WakefulIntentService

For our trivial sample, where we are merely logging to LogCat, we could simply move that logic out of an `IntentService` and into a `BroadcastReceiver`. Then, Android would ensure that the device would stay awake long enough for us to do our work in `onReceive()`.

The problem is that `onReceive()` is called on the main application thread, so we cannot spend much time in that method. And, since our alarm event might occur when nothing else of our code is running, we need to have our `BroadcastReceiver` registered in the manifest, rather than via `registerReceiver()`. A side effect of this is that we cannot fork threads or do other things in `onReceive()` that might live past `onReceive()` yet be "owned" by the `BroadcastReceiver` itself. Besides, Android only ensures that the device will stay awake until `onReceive()` returns, so even if we did fork a thread, the device might fall asleep before that thread can complete its work.

Enter the `WakefulIntentService`.

`WakefulIntentService` is a reusable component, published by the author of this book.

`WakefulIntentService` allows you to implement "the handoff pattern":

- You add the JAR, AAR, or library project to your project
- You create a subclass of `WakefulIntentService` to do your background work, putting that business logic in a `doWakefulWork()` method instead of `onHandleIntent()` (though it is still called on a background thread)
- You set up your alarm to route to a `BroadcastReceiver` of your design
- Your `BroadcastReceiver` calls `sendWakefulWork()` on the `WakefulIntentService` class, identifying your own subclass of `WakefulIntentService`
- You add a `WAKE_LOCK` permission to your manifest

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`WakefulIntentService` will perform a bit of magic to ensure that the device will stay awake long enough for your work to complete in `doWakefulWork()`. Hence, we get the best of both worlds: the device will not fall asleep, and we will not have to worry about tying up the main application thread.

Android Studio users can add a reference to the CommonsWare Maven artifact repository in their top-level `repositories` closure:

```
repositories {
    mavenCentral()
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}
```

(here shown alongside an existing `mavenCentral()` statement)

Then, adding the `WakefulIntentService` is merely a matter of adding a `compile 'com.commonsware.cwac:wakeful:...'` statement to the top-level `dependencies` closure (for some version of the library, denoted by `...`).

Eclipse users can download a JAR from the project's GitHub repository.

`WakefulIntentService` is open source, licensed under the Apache License 2.0.

The Polling Archetype, Revisited

With that in mind, take a peek at the `AlarmManager/Wakeful` sample project. This is a near-clone of the previous sample, with the primary difference being that we will use `WakefulIntentService`.

The `libs/` directory of the project contains the `CWAC-WakefulIntentService.jar` library, so Eclipse/Ant users can make use of `WakefulIntentService` in our code. Android Studio users, instead, pull the AAR from the CommonsWare artifact repository:

```
repositories {
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}

dependencies {
    compile 'com.commonsware.cwac:wakeful:1.0.+'
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Our manifest includes the WAKE_LOCK permission:

```
  <uses-permission android:name="android.permission.WAKE_LOCK"/>
```

Our PollReceiver will now serve two roles: handling ACTION_BOOT_COMPLETED *and* handling our alarm events. We can detect which of these cases triggered onReceive() by inspecting the broadcast Intent, passed into onReceive(). We will use an explicit Intent for the alarm events, so any Intent with an action string must be ACTION_BOOT_COMPLETED:

```
package com.commonsware.android.wakesvc;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class PollReceiver extends BroadcastReceiver {
  private static final int PERIOD=900000; // 15 minutes
  private static final int INITIAL_DELAY=5000; // 5 seconds

  @Override
  public void onReceive(Context ctxt, Intent i) {
    if (i.getAction() == null) {
      WakefulIntentService.sendWakefulWork(ctxt, ScheduledService.class);
    }
    else {
      scheduleAlarms(ctxt);
    }
  }

  static void scheduleAlarms(Context ctxt) {
    AlarmManager mgr=
        (AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE);
    Intent i=new Intent(ctxt, PollReceiver.class);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);

    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                     SystemClock.elapsedRealtime() + INITIAL_DELAY,
                     PERIOD, pi);

  }
}
```

If the Intent is our explicit Intent, we call sendWakefulWork() on WakefulIntentService, identifying our ScheduledService class as being the service that contains our business logic.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The other changes to `PollReceiver` is that we use `getBroadcast()` to create our `PendingIntent`, wrapping our explicit `Intent` identifying `PollReceiver` itself, and that we use more realistic polling periods (5 second initial delay, every 15 minutes thereafter).

`ScheduledService` has only two changes: it extends `WakefulIntentService` and has the LogCat logging in `doWakefulWork()`:

```
package com.commonsware.android.wakesvc;

import android.content.Intent;
import android.util.Log;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class ScheduledService extends WakefulIntentService {
  public ScheduledService() {
    super("ScheduledService");
  }

  @Override
  protected void doWakefulWork(Intent intent) {
    Log.d(getClass().getSimpleName(), "I ran!");
  }
}
```

How the Magic Works

A `WakefulIntentService` keeps the device awake by using a `WakeLock`. A `WakeLock` allows a "userland" (e.g., Android SDK) app to tell the Linux kernel at the heart of Android to keep the device awake, with the CPU powered on, indefinitely, until the `WakeLock` is released.

This can be a wee bit dangerous, as you can accidentally keep the device awake much longer than you need to. That is why using a library like `WakefulIntentService` can be useful — to use more-tested code rather than rolling your own.

Warning: Not All Android Devices Play Nice

Some Android devices take liberties with the way `AlarmManager` works, in ways that may affect your applications.

One example of this today is the SONY Xperia Z. It has a "STAMINA mode" that the user can toggle on via the "Power Management" screen in Settings. This mode will

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

be entered when the device's screen turns off, if the device is not plugged in and charging. The user can add apps to a whitelist ("Apps active in standby"), where STAMINA mode does not affect those apps' behavior.

`_WAKEUP` style alarms do not wake up the device when it is in STAMINA mode. The behavior is a bit reminiscent of non-`_WAKEUP` alarms. Alarms that occur while the device is asleep are suppressed, and you get one invocation of your `PendingIntent` at the point the device wakes back up. At that point, the schedule continues as though the alarms had been going off all along. Apps on the whitelist are unaffected.

Mostly, you need to be aware of this from a support standpoint. If Xperia Z owners complain that your app behaves oddly, and you determine that your alarms are not going off, see if they have STAMINA mode on, and if they do, ask them to add your app to the whitelist.

If you are using "if my alarm has not gone off in X amount of time, the user perhaps force-stopped me, so let me reschedule my alarms" logic, you should be OK. Before one of your activities gets a chance to make that check, your post-wakeup alarm should have been invoked, so you can update your event log and last-run timestamp. Hence, you should not be tripped up by STAMINA and accidentally reschedule your alarms (potentially causing duplicates, depending upon your alarm-scheduling logic).

Other devices with similar characteristics include Sony's Xperia P, Xperia U, Xperia sola, and Xperia go.

Debugging Alarms

If you are encountering issues with your alarms, the first thing to do is to ensure that the alarm schedule in `AlarmManager` is what you expect it to be. To do that, run `adb shell dumpsys alarm` from a command prompt. This will dump a report of all the scheduled alarms, including when they are set to be invoked next (with portions replaced by vertical ellipses to keep this listing from being too long):

```
Current Alarm Manager state:

  Realtime wakeup (now=2013-03-09 07:49:51):
  RTC_WAKEUP #11: Alarm{429c6028 type 0 com.android.providers.calendar}
    type=0 when=+21h40m9s528ms repeatInterval=0 count=0
    operation=PendingIntent{42ec2f40: PendingIntentRecord{434fb2f8
com.android.providers.calendar broadcastIntent}}
  RTC_WAKEUP #10: Alarm{42e17e28 type 0 com.google.android.gms}
    type=0 when=+18h10m8s480ms repeatInterval=86400000 count=1
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    operation=PendingIntent{42e15d20: PendingIntentRecord{42e0cc28
com.google.android.gms startService}}
.
.
.

  Elapsed realtime wakeup (now=+6d15h50m2s672ms):
  ELAPSED_WAKEUP #16: Alarm{42cf26f0 type 2 com.google.android.apps.maps}
    type=2 when=+999d23h59m59s999ms repeatInterval=0 count=0
    operation=PendingIntent{42de2dc0: PendingIntentRecord{42ac73e8
com.google.android.apps.maps broadcastIntent}}
  ELAPSED_WAKEUP #15: Alarm{42c4a638 type 2 com.google.android.apps.maps}
    type=2 when=+1d18h10m8s894ms repeatInterval=0 count=0
    operation=PendingIntent{42ab50c8: PendingIntentRecord{42e2c020
com.google.android.apps.maps broadcastIntent}}
.
.
.

  Broadcast ref count: 0

  Top Alarms:
    +14m24s97ms running, 0 wakeups, 9567 alarms: android
       act=android.intent.action.TIME_TICK
    +1m15s72ms running, 4890 wakeups, 4890 alarms: com.android.phone
       act=com.android.server.sip.SipWakeupTimer@42626830
    +1m13s465ms running, 0 wakeups, 320 alarms: android
       act=com.android.server.action.NETWORK_STATS_POLL
    +45s803ms running, 0 wakeups, 639 alarms: com.google.android.deskclock
       act=com.android.deskclock.ON_QUARTER_HOUR
    +42s830ms running, 0 wakeups, 19 alarms: com.android.phone
       act=com.android.phone.UPDATE_CALLER_INFO_CACHE cmp={com.android.phone/
com.android.phone.CallerInfoCacheUpdateReceiver}
    +35s479ms running, 0 wakeups, 954 alarms: android
       act=com.android.server.ThrottleManager.action.POLL
    +14s28ms running, 1609 wakeups, 1609 alarms: com.android.phone
       act=com.android.internal.telephony.gprs-data-stall
    +11s98ms running, 171 wakeups, 171 alarms: com.android.providers.calendar
       act=com.android.providers.calendar.intent.CalendarProvider2
    +8s380ms running, 893 wakeups, 893 alarms: android
       act=android.content.syncmanager.SYNC_ALARM
    +8s353ms running, 569 wakeups, 569 alarms: com.google.android.apps.maps
       cmp={com.google.android.apps.maps/
com.google.googlenav.prefetch.android.PrefetcherService}

  Alarm Stats:
  com.google.android.location +120ms running, 12 wakeups:
    +73ms 7 wakes 7 alarms:
act=com.google.android.location.nlp.ALARM_WAKEUP_CACHE_UPDATER
    +47ms 5 wakes 5 alarms:
act=com.google.android.location.nlp.ALARM_WAKEUP_LOCATOR
  android +15m32s920ms running, 1347 wakeups:
    +14m24s97ms 0 wakes 9567 alarms: act=android.intent.action.TIME_TICK
    +1m13s465ms 0 wakes 320 alarms:
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
act=com.android.server.action.NETWORK_STATS_POLL
    +35s479ms 0 wakes 954 alarms:
act=com.android.server.ThrottleManager.action.POLL
    +8s380ms 893 wakes 893 alarms: act=android.content.syncmanager.SYNC_ALARM
    +7s734ms 159 wakes 159 alarms: act=android.appwidget.action.APPWIDGET_UPDATE
cmp={com.guywmustang.silentwidget/
com.guywmustang.silentwidgetlib.SilentWidgetProvider}
    +1s144ms 151 wakes 151 alarms: act=android.app.backup.intent.RUN
    +922ms 0 wakes 6 alarms: act=android.intent.action.DATE_CHANGED
    +479ms 66 wakes 66 alarms:
act=com.android.server.WifiManager.action.DEVICE_IDLE
    +383ms 56 wakes 56 alarms:
act=com.android.server.WifiManager.action.DELAYED_DRIVER_STOP
    +101ms 14 wakes 14 alarms:
act=com.android.server.action.UPDATE_TWILIGHT_STATE
    +100ms 7 wakes 7 alarms:
act=com.android.internal.policy.impl.PhoneWindowManager.DELAYED_KEYGUARD
    +9ms 1 wakes 1 alarms: act=android.net.wifi.DHCP_RENEW
    +3ms 0 wakes 1 alarms:
act=com.android.server.NetworkTimeUpdateService.action.POLL
  com.google.android.apps.maps +14s742ms running, 911 wakeups:
    +8s353ms 569 wakes 569 alarms: cmp={com.google.android.apps.maps/
com.google.googlenav.prefetch.android.PrefetcherService}
    +2s211ms 85 wakes 85 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_LOCATOR
    +1s206ms 103 wakes 103 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_SENSOR_UPLOADER
    +807ms 2 wakes 2 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_BURST_COLLECTION_TRIGGER
    +759ms 56 wakes 56 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_S_COLLECTOR
    +566ms 10 wakes 10 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_CACHE_UPDATER
    +385ms 39 wakes 39 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_IN_OUT_DOOR_COLLECTOR
    +308ms 31 wakes 31 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_ACTIVE_COLLECTOR
    +77ms 8 wakes 8 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_ACTIVITY_DETECTION
    +42ms 4 wakes 4 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_PASSIVE_COLLECTOR
    +28ms 4 wakes 4 alarms:
act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_CALIBRATION_COLLECTOR
.
.
.
```

You are given details of each outstanding alarm, including the all-important `when` value indicating the time the alarm should be invoked next, if it is not canceled first (e.g., `when=+5d15h10m7s782ms`), along with the package requesting the alarm. You can use this to identify your app's alarms and see when they should be invoked next.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You are also given:

- Per-app details about how frequently their alarms have gone off, which can be useful for battery impact analysis
- A list of "top alarms" by number of occurrences, also for device performance analysis

Note, though, that for inexact alarms, the `when` value may not indicate when the event will actually occur.

WakefulBroadcastReceiver

The Android Support package has added a `WakefulBroadcastReceiver`, which offers an alternative to `WakefulIntentService` for arranging to do work, triggered by a broadcast (such as an `AlarmManager` event), that may take a while. `WakefulBroadcastReceiver` has its pros and cons compared to `WakefulIntentService`, making it worth considering.

Using WakefulBroadcastReceiver

Using `WakefulBroadcastReceiver` with `AlarmManager` is slightly different than is using `WakefulIntentService`. The `AlarmManager/WakeCast` sample project is a clone of the `WakefulIntentService` project, but one using `WakefulBroadcastReceiver` instead.

The activity is unchanged — it simply calls `scheduleAlarms()` on `PollReceiver`. `scheduleAlarms()` itself is unchanged, as it still uses `setRepeating()` on `AlarmManager` to arrange to periodically invoke a `PendingIntent`, targeting the `PollReceiver`.

But `PollReceiver` itself is now a `WakefulBroadcastReceiver` rather than just an ordinary `BroadcastReceiver`. This in turn requires a slightly different implementation of `onReceive()`:

```
package com.commonsware.android.wakecast;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;
import android.support.v4.content.WakefulBroadcastReceiver;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public class PollReceiver extends WakefulBroadcastReceiver {
  private static final int PERIOD=900000; // 15 minutes
  private static final int INITIAL_DELAY=5000; // 5 seconds

  @Override
  public void onReceive(Context ctxt, Intent i) {
    if (i.getAction() == null) {
      startWakefulService(ctxt,
                          new Intent(ctxt, ScheduledService.class));
    }
    else {
      scheduleAlarms(ctxt);
    }
  }

  static void scheduleAlarms(Context ctxt) {
    AlarmManager mgr=
        (AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE);
    Intent i=new Intent(ctxt, PollReceiver.class);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);

    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                     SystemClock.elapsedRealtime() + INITIAL_DELAY,
                     PERIOD, pi);

  }
}
```

Now, when the `AlarmManager` broadcast arrives, we call `startWakefulService()`, passing it the `Context` supplied to `onReceive()`, plus an `Intent` identifying the service to start up. Under the covers, this works much like `sendWakefulWork()` on `WakefulIntentService` — it starts the identified service, but acquires a `WakeLock` first.

Our `ScheduledService` is now a regular `IntentService`, instead of a `WakefulIntentService`. This means that our background work moves back to the standard `onHandleIntent()` method, instead of `doWakefulWork()`. However, we have one extra bit of bookkeeping to do: we must call the static `completeWakefulIntent()` method on `WakefulBroadcastReceiver` (or, as shown, on `PollReceiver`, as that will point to the same static method):

```
package com.commonsware.android.wakecast;

import android.app.IntentService;
import android.content.Intent;
import android.util.Log;

public class ScheduledService extends IntentService {
  public ScheduledService() {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super("ScheduledService");
  }

  @Override
  protected void onHandleIntent(Intent intent) {
    Log.d(getClass().getSimpleName(), "I ran!");

    PollReceiver.completeWakefulIntent(intent);
  }
}
```

We pass the `Intent` supplied to `onHandleIntent()` to `completeWakefulIntent()`. Behind the scenes, `completeWakefulIntent()` will release the `WakeLock` that has been keeping our CPU powered on while we do our work.

Comparing to WakefulIntentService

One might think that `WakefulIntentService` would now be obsolete with the addition of `WakefulBroadcastReceiver`. In truth, there are some advantages to the current implementation of `WakefulBroadcastReceiver`:

- It uses a time-limited `WakeLock`, one set to auto-release after one minute, so there is no risk of an app somehow failing to release the lock and thereby keeping the CPU on indefinitely.
- To make the time-limited locks work, `WakefulBroadcastReceiver` uses one `WakeLock` per request, rather than the single static `WakeLock` that `WakefulIntentService` uses, making `WakefulBroadcastReceiver` incrementally more resilient in the face of various potential problems.
- Because it is not strictly tied to being used with an `IntentService`, `WakefulBroadcastReceiver` may offer greater flexibility. For example, an `IntentService` is not a good choice if the work you do is intrinsically asynchronous, such as trying to find the device's location. Any place where you find yourself registering a listener from a service, an `IntentService` will not work well, as the `IntentService` wants to shut down before your listener has received a result. A regular `Service` can work well, though, in this case, and `WakefulBroadcastReceiver` might be of use in this pattern (though the author has not tried this yet).

On the other hand:

- `WakefulBroadcastReceiver` requires an explicit call to `completeWakefulIntent()`, which a developer can easily forget, possibly causing the `WakeLock` to be leaked. While this is not disastrous, since the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`WakeLock` will auto-release after a minute, it may still represent wasted power. `WakefulIntentService` is more "idiot-proof" and therefore avoids this issue.

- The time for the `WakefulBroadcastReceiver` `WakeLock` is locked to being one minute — no more, no less. This offers limited flexibility and can cause problems if the work you intend to do could easily exceed a minute. Unfortunately, the implementation of `WakefulBroadcastReceiver` offers no easy way to override this one-minute timeout value.
- If Android terminates your process and restarts your service, the restarted service will *not* be under the control of a `WakeLock`, as Android will be starting the service directly, not via `WakefulBroadcastReceiver`. `WakefulIntentService` will suffer the same fate, but it will automatically grab a `WakeLock` for you when it detects this condition. In the case of `WakefulBroadcastReceiver`, your service will run without a `WakeLock`, unless you detect this case yourself (via a custom `onStartCommand()` that examines the passed-in `flags`, looking for `START_FLAG_REDELIVERY`) and grab your own `WakeLock`.

A future generation of `WakefulIntentService` will aim to adopt some of the advantages of `WakefulBroadcastReceiver` while avoiding its disadvantages. As it stands, either component is a reasonable choice if you are willing to live within their respective constraints.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #17 - Periodic Book Updates

Now that we have the ability to update our book's prose by downloading some files from a Web site, we can take the next step: update the book automatically, on a scheduled basis.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Adding a Stub UpdateReceiver

This tutorial is going to use `AlarmManager`. Therefore, we will need a manifest-registered `BroadcastReceiver`, for two reasons:

1. We need to get control at boot time, to restore our alarm schedule
2. We need something to get control when the alarm events occur

In this step, to solve both needs, we will set up a stub `UpdateReceiver`.

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio

Right-click over the `com.commonsware.empublite` package in your `java/` directory and choose New > Other > "Broadcast Receiver" from the context menu, to bring up the new-`BroadcastReceiver` dialog:

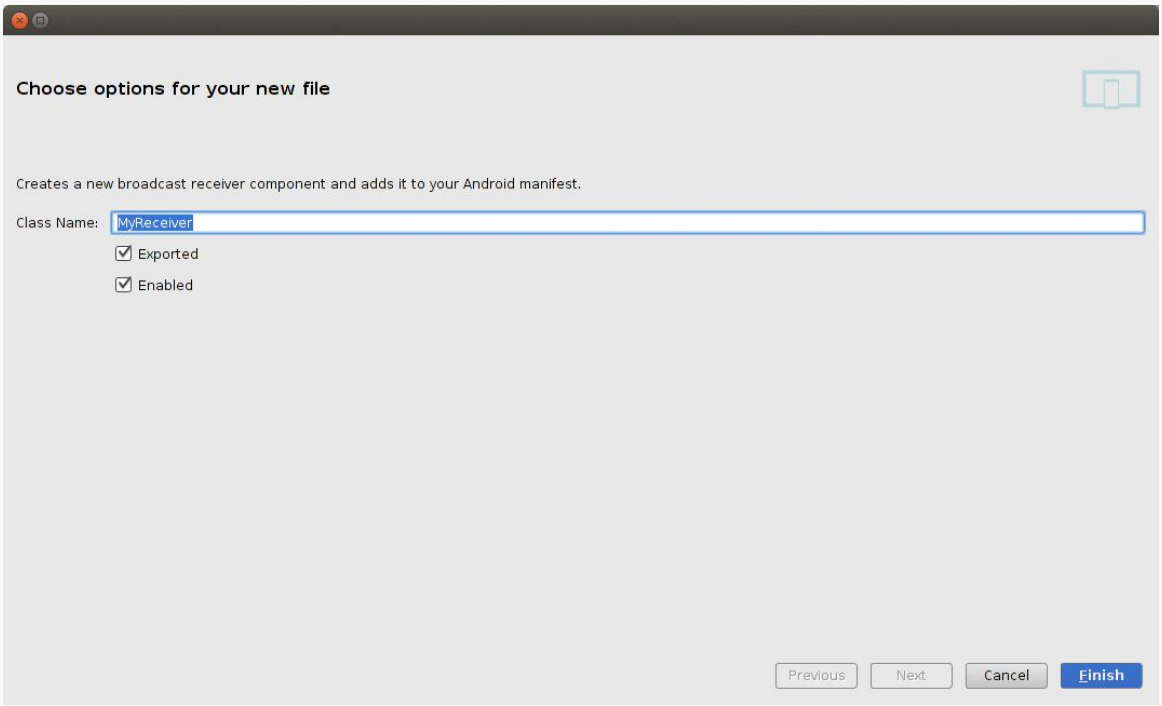


Figure 289: Android Studio New BroadcastReceiver Dialog

Fill in `UpdateReceiver` as the class name. Make sure that the Exported and Enabled checkboxes are checked. Then, click Finish to generate the `UpdateReceiver` class and add an entry for you to the manifest.

Update the generated `<receiver>` element in the manifest to look like:

```
    <receiver android:name="UpdateReceiver">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>
    </receiver>
```

Also, add the following `<uses-permission>` element alongside the other such elements in the manifest:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
```

Finally, in the generated `UpdateReceiver` class, get rid of the completely incorrect zero-argument constructor that is erroneously code-generated for you.

Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. Fill in `UpdateReceiver` in the "Name" field. Click the "Browse..." button next to the "Superclass" field and find `BroadcastReceiver` to set as the superclass. Then, click "Finish" on the new-class dialog to create the `UpdateReceiver` class.

You will also need to add a new receiver node to the list of nodes in the Application sub-tab of `AndroidManifest.xml`, pointing to `UpdateReceiver`, following the same approach that we used for other receivers in this application.

However, we also must add an `<intent-filter>` to the `<receiver>` element, identifying the broadcast which we wish to monitor. To do that:

- Click on the Receiver element associated with `UpdateReceiver` in the list of "Application Nodes"
- Click the "Add..." button next to the list of "Application Nodes" and choose "Intent Filter" from the list
- With the "Intent Filter" highlighted in the "Application Nodes" tree, click "Add..." again, this time choosing "Action" from the list
- In the details area on the right, choose `android.intent.action.BOOT_COMPLETED`

Also, in the Permissions sub-tab of the manifest editor, you will need to add the `RECEIVE_BOOT_COMPLETED` "Uses Permission" node.

Step #2: Scheduling the Alarms

Somewhere, we need code to schedule the alarms with `AlarmManager`. Ideally, this will be a static method, one we can use from both `EmPubLiteActivity` (for normal scheduling) and `UpdateReceiver` (for scheduling at boot time).

With that in mind, add the following `scheduleAlarm()` static method to `UpdateReceiver`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
static void scheduleAlarm(Context ctxt) {
  AlarmManager mgr=
      (AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE);
  Intent i=new Intent(ctxt, UpdateReceiver.class);
  PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);
  Calendar cal=Calendar.getInstance();

  cal.set(Calendar.HOUR_OF_DAY, 4);
  cal.set(Calendar.MINUTE, 0);
  cal.set(Calendar.SECOND, 0);
  cal.set(Calendar.MILLISECOND, 0);

  if (cal.getTimeInMillis() < System.currentTimeMillis()) {
    cal.add(Calendar.DAY_OF_YEAR, 1);
  }

  mgr.setRepeating(AlarmManager.RTC_WAKEUP, cal.getTimeInMillis(),
                   AlarmManager.INTERVAL_DAY, pi);
}
```

Here we create a broadcast `PendingIntent` pointing back at `UpdateReceiver`, create a `Calendar` object for tomorrow at 4am, and call `setRepeating()` on `AlarmManager` to invoke our `PendingIntent` every day at 4am.

Then, modify `onReceive()` of `UpdateReceiver` to use `scheduleAlarm()`, if we are called with an action string (indicating that we are being called due to `ACTION_BOOT_COMPLETED`):

```
@Override
public void onReceive(Context context, Intent intent) {
  if (intent.getAction() != null) {
    scheduleAlarm(context);
  }
}
```

Finally, at the end of `onCreate()` of `EmPubLiteActivity`, add:

```
    UpdateReceiver.scheduleAlarm(this);
```

This will schedule the alarms whenever the app is run. Between that and `UpdateReceiver`, the alarms should be active most of the time during normal operation.

Step #3: Adding the WakefulIntentService

It is possible that at 4am local time — or whenever the alarm goes off — that the user will not be using their device. Therefore, it is possible that the device will fall

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

asleep while we try to download the update. Therefore, we need to switch to using `WakefulIntentService`.

Android Studio users can adjust their top-level `repositories` closure in `app/build.gradle` to be:

```
repositories {
    mavenCentral()
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}
```

Then, Android Studio users can add the following line to the `dependencies` closure, to pull in the CWAC-Wakeful AAR artifact from the CommonsWare repository:

```
compile 'com.commonsware.cwac:wakeful:1.0.+'
```

If you get the yellow banner asking you to do a project sync, please do so.

Eclipse users should visit the `WakefulIntentService` releases and download the latest JAR file listed there. Put it in the `libs/` directory of your project — you can either:

- Do this work inside of Eclipse (e.g., drag-and-drop the JAR into Package Explorer), or
- Do this work outside of Eclipse (e.g., create the `libs/` directory directly using OS tools), then press `<F5>` over the project to get Eclipse to scan the project's directory and pick up your changes

Users of either IDE can then modify `DownloadCheckService` to inherit from `com.commonsware.cwac.wakeful.WakefulIntentService` instead of from `IntentService`. This will cause you to need to rename your `onHandleIntent()` method to be `doWakefulWork()`.

Also, add the `WAKE_LOCK` permission in the manifest, along with the rest of our permissions. Eclipse users can add this from the Permissions sub-tab of the Eclipse manifest editor; non-Eclipse users can add another `<uses-permission>` element:

```
<uses-permission android:name="android.permission.WAKE_LOCK"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Step #4: Using WakefulIntentService

To correctly use `WakefulIntentService`, we need to use `sendWakefulWork()` to send commands to one, rather than `startService()`.

With that in mind, in `EmPubLiteActivity`, change the `R.id.update` `case` of the `switch` statement in `onOptionsItemSelected()` to use `sendWakefulWork()`:

```
      case R.id.update:
        WakefulIntentService.sendWakefulWork(this, DownloadCheckService.class);

        return(true);
```

Step #5: Completing the UpdateReceiver

Finally, add an `else` block to the `if` statement in `onReceive()` of `UpdateReceiver`, to handle the case where we get control due to the alarm event, so we can use `sendWakefulWork()` to invoke the `DownloadCheckService`:

```
@Override
public void onReceive(Context context, Intent intent) {
  if (intent.getAction() != null) {
    scheduleAlarm(context);
  }
  else {
    WakefulIntentService.sendWakefulWork(context,
                                         DownloadCheckService.class);
  }
}
```

To test this:

1. Uninstall the existing EmPubLite application (e.g., by using the Settings app)
2. Install and run the revised app in your device or emulator, confirm that you are viewing the non-updated book, then press BACK to exit the activity
3. Temporarily modify the time of your emulator to be a few minutes before 4am either today (if the current time is between midnight and 4am) or tomorrow (if the current time is after 4am)
4. Return EmPubLite to the foreground (due to a bug in the current tutorial that will be fixed sometime in 2015...)
5. Find something to pass the time for the rest of those few minutes, such as procuring liquid refreshment suitable for the time of day and locale

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

6. A few minutes after the simulated 4am, confirm that you have downloaded the updated book (check the first page of the `ViewPager`), then fix your emulator's clock back to normal

In Our Next Episode…

… we will let the user know about updates from the background via a `Notification`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Notifications

Pop-up messages. Tray icons and their associated "bubble" messages. Bouncing dock icons. You are no doubt used to programs trying to get your attention, sometimes for good reason.

Your phone also probably chirps at you for more than just incoming calls: low battery, alarm clocks, appointment notifications, incoming text message or email, etc.

Not surprisingly, Android has a whole framework for dealing with these sorts of things, collectively called "notifications".

What's a Notification?

A service, running in the background, needs a way to let users know something of interest has occurred, such as when email has been received. Moreover, the service may need some way to steer the user to an activity where they can act upon the event – reading a received message, for example. For this, Android supplies status bar icons, flashing lights, and other indicators collectively known as "notifications".

Your current phone may well have such icons, to indicate battery life, signal strength, whether Bluetooth is enabled, and the like. With Android, applications can add their own status bar icons, with an eye towards having them appear only when needed (e.g., a message has arrived).

Notifications will appear in one of two places. On most devices, they will appear in the status bar, on the top of the screen, left-aligned:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

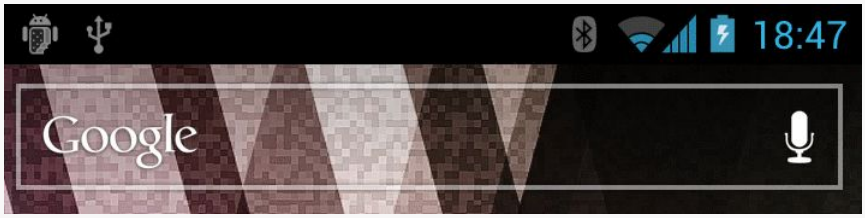


Figure 290: Notifications, on a Galaxy Nexus

On a pre-Android 4.2 tablet (and occasionally on other tablets newer than that), they will appear in the system bar, on the bottom of the screen, towards the lower-right corner:

Figure 291: Notifications, on a Galaxy Tab 2

In either case, you can expand the "notification drawer" to get more details about the active notifications, either by sliding down the status bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

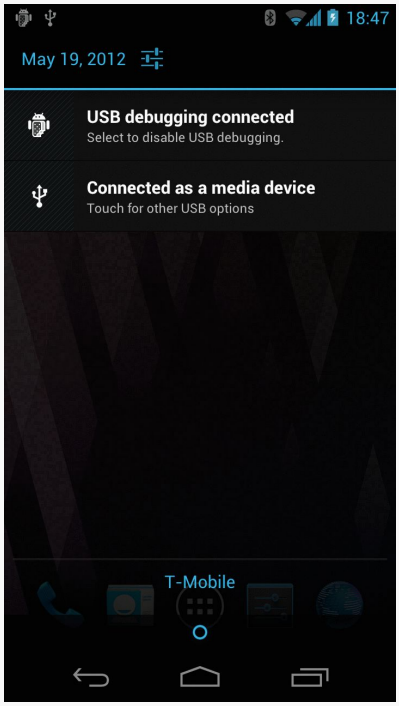


Figure 292: Notification Drawer, on a Galaxy Nexus

or by tapping on the clock on the system bar on some tablets:

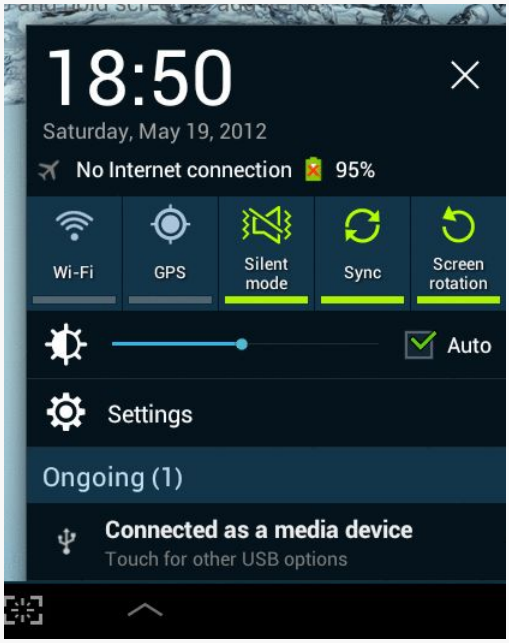


Figure 293: Notification Drawer, on a Galaxy Tab 2

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Some notifications will be complex, showing real-time information, such as the progress of a long download. More often, notifications are fairly simple, providing just a couple of lines of information, plus an identifying icon. Tapping on the notification drawer entry will typically trigger some action, such as starting an activity — an email app letting the user know that "you've got mail" can have its notification bring up the inbox activity when tapped.

Showing a Simple Notification

Previously in the book, we had an example of using `DownloadManager`. There, we would let the user know about the completion of our download by sending a broadcast `Intent` back to the activity, so it could do something — in our case, display a `Toast`.

An alternative would be for the background service doing the download to raise a `Notification` when the download is complete. That would work even if the activity was no longer around (e.g., user pressed BACK to exit it). A modified version of the original `DownloadManager` sample taking this `Notification` approach can be found in the `Notifications/DownloadNotify` sample project.

Our `DownloadFragment` for triggering the download has two changes:

1. We dispense with the `BroadcastReceiver` and logic related to it, including disabling and enabling the `Button`
2. On the `Intent` we use with `startService()`, we include not only the `Uri` of the file to download, but also its MIME type, by calling `setDataAndType()` on the `Intent` object

```
package com.commonsware.android.downloader;

import android.app.Fragment;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;

public class DownloadFragment extends Fragment implements
    View.OnClickListener {
  private Button b=null;

  @Override
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.main, parent, false);

    b=(Button)result.findViewById(R.id.button);
    b.setOnClickListener(this);

    return(result);
  }

  @Override
  public void onClick(View v) {
    Intent i=new Intent(getActivity(), Downloader.class);

    i.setDataAndType(Uri.parse("http://commonsware.com/Android/excerpt.pdf"),
                     "application/pdf");

    getActivity().startService(i);
    getActivity().finish();
  }
}
```

The download logic in the `onHandleIntent()` method of `Downloader` is nearly identical. The difference is that at the end, rather than sending a broadcast `Intent`, we call a private `raiseNotification()` method. We also call this method if there is an exception during the download. The `raiseNotification()` method takes the `Intent` command that was delivered to `onHandleIntent()`, the `File` object representing the downloaded results (if we succeeded), and the `Exception` that was raised (if we crashed). As one might guess given the method's name, `raiseNotification()` will raise a `Notification`:

```
  private void raiseNotification(Intent inbound, File output,
                                 Exception e) {
    NotificationCompat.Builder b=new NotificationCompat.Builder(this);

    b.setAutoCancel(true).setDefaults(Notification.DEFAULT_ALL);

    if (e == null) {
      b.setContentTitle(getString(R.string.download_complete))
       .setContentText(getString(R.string.fun))
       .setSmallIcon(android.R.drawable.stat_sys_download_done)
       .setTicker(getString(R.string.download_complete));

      Intent outbound=new Intent(Intent.ACTION_VIEW);

      outbound.setDataAndType(Uri.fromFile(output), inbound.getType());

      b.setContentIntent(PendingIntent.getActivity(this, 0, outbound, 0));
    }
    else {
      b.setContentTitle(getString(R.string.exception))
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      .setContentText(e.getMessage())
      .setSmallIcon(android.R.drawable.stat_notify_error)
      .setTicker(getString(R.string.exception));
  }

  NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

  mgr.notify(NOTIFY_ID, b.build());
}
```

The first thing we do in `raiseNotification()` is create a `Builder` object to help construct the `Notification`. On API Level 11 and higher, there is a `Notification.Builder` class that you can use.

However, the notification system in Android has been changing frequently over the past few OS updates, and there are signs that this will continue. Hence, you may prefer to use `NotificationCompat.Builder`. First, this will work back to API Level 4, in case you are supporting notifications on older devices. More importantly, `NotificationCompat.Builder` is updated to reflect the latest `Notification.Builder` API, offering a backwards-compatible implementation of that API. Some newer features are not supported on older devices, but the `NotificationCompat.Builder` API lets you code to the new API, and it quietly ignores things that cannot be done on older devices.

We can call methods on the `Builder` to configure the `Notification` that we want to display. Whether our download succeeded or failed, we use various methods on `Builder`:

- `setAutoCancel(true)` means that when the user slides open the notification drawer and taps on our entry, the `Notification` is automatically canceled and goes away
- `setDefaults(Notification.DEFAULT_ALL)` means that we want the device's standard notification tone, LED light flash, and vibration to occur when the `Notification` is displayed

If we succeeded (the passed-in `Exception` is `null`), we further configure our `Notification` via more calls to the `Builder`:

- `setContentTitle()` and `setContentText()` supply the prose to display in the two lines of the notification drawer entry for our `Notification`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `setSmallIcon()` indicates the icon to display in the status bar or system bar when the `Notification` is active (in this case, specifying one supplied by Android itself)
- `setTicker()` supplies some text to be displayed in the status bar or system bar for a few seconds right when the `Notification` is displayed, so users who happen to be looking at their device at that time will get more information at a glance about what just happened that is demanding their attention

In addition, `setContentIntent()` supplies a `PendingIntent` to be invoked when the notification drawer entry for our `Notification` is tapped. In our case, we create an `ACTION_VIEW` `Intent` for our `File` (using `Uri.fromFile()` to get a `Uri` pointing to our file on external storage) with the MIME type supplied from `DownloadFragment`. Hence, if the user taps on our notification drawer entry, we will attempt to bring up a PDF viewer on the downloaded PDF file – whether this will succeed or not will depend upon whether there is a PDF viewer installed on the device.

If, instead, we did have an `Exception`, we use the same methods on `Builder` (minus `setContentIntent()`) to configure the `Notification`, but using different text and icons.

To actually display the `Notification`, we need to get a `NotificationManager`, which is another system service. Calling `getSystemService()` and asking for the `NOTIFICATION_SERVICE` will give us our `NotificationManager`, albeit after a cast. Then, we can call `notify()` on the `NotificationManager`, supplying our `Notification` (from `build()` on the `Builder`) and a locally-unique integer (`NOTIFY_ID`, defined as a static data member on the service). That integer can later be used with a `cancel()` method to remove the `Notification` from the screen, even if the user has not canceled it themselves (e.g., via tapping on it with `setAutoCancel(true)`).

NOTE: You may see some samples using `getNotification()` with `NotificationBuilder` instead of `build()`. `getNotification()` was the original method, but it has since been deprecated in favor of `build()`.

Also, because we are using `setDefaults(Notification.DEFAULT_ALL)`, and since the default behavior for a `Notification` may involve vibrating the phone, we need to hold the `VIBRATE` permission in the manifest:

```
  <uses-permission android:name="android.permission.VIBRATE"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that as of Android 4.4, you no longer need the `VIBRATE` permission, if you are using `DEFAULT_ALL` or `DEFAULT_VIBRATE` for `setDefaults()`. If you specify a custom vibration pattern, via `setVibrate()`, you still need the `VIBRATE` permission.

Running this in a device or emulator will display the `Notification` upon completion of the download:

Figure 294: Sample Notification, on Android 4.4

Opening the notification drawer displays our `Notification` details:

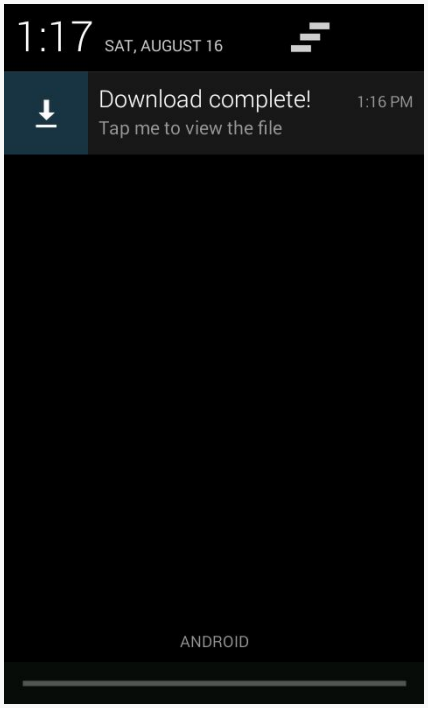


Figure 295: Sample Notification in Drawer, on a Galaxy Nexus

Tapping on the drawer entry will try to start a PDF viewer, perhaps bringing up a chooser if there are multiple such viewers on the device. Also, tapping on the drawer entry will cancel the `Notification` and remove it from the screen.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Activity-Or-Notification Scenario

Let us suppose that you are writing an email app. In addition to an "inbox" activity, you have an `IntentService`, scheduled via `AlarmManager`, to go check for new email messages every so often. This means, when your service discovers and downloads new messages, there are two possibilities:

- The user has your inbox activity in the foreground, and that activity should update to reflect the fact that there are new messages
- The user does *not* have your inbox activity in the foreground, so you want to display a `Notification` to alert the user of the new messages and lead them back to the inbox

However, ideally, the service neither knows nor cares whether the inbox activity is in the foreground, exists in the process but is not in the foreground, or does not exist in the process (e.g., Android started a new process to handle this middle-of-the-night check for new email messages).

One way to handle this is via an event bus.

The recipe for the `Activity-or-Notification` pattern is:

1. Define an event (e.g., event class for greenrobot's EventBus, custom action string for `LocalBroadcastManager`)
2. Have your activity or fragment register to respond to these events while in the foreground (e.g., in `onResume()`) and unregister when leaving the foreground (e.g., `onPause()`). The activity or fragment can then update the UI in response to the event.
3. The service raises the event bus event at appropriate times.
4. By some means appropriate to the event bus implementation, the service needs to know whether an activity or fragment responded to the event, so it can raise a `Notification` if the event has not already been handled.

We will see three implementations of this pattern in the chapter on event bus alternatives.

Big (and Rich) Notifications

Android 4.1 introduced new `Notification` styles that automatically expand into a "big" area when they are the top `Notification` in the drawer. These expanded

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Notifications can display more text (or a larger thumbnail of an image), plus add some action buttons to allow the user to directly perform more actions straight from the Notification itself.

And while these new Notification styles are only available on API Level 16 and higher, a familiar face has created a compatibility layer so our code can request the larger styles and still work on older devices.

The Styles

There are three main styles supplied for expanded Notifications. There is the BigText style:

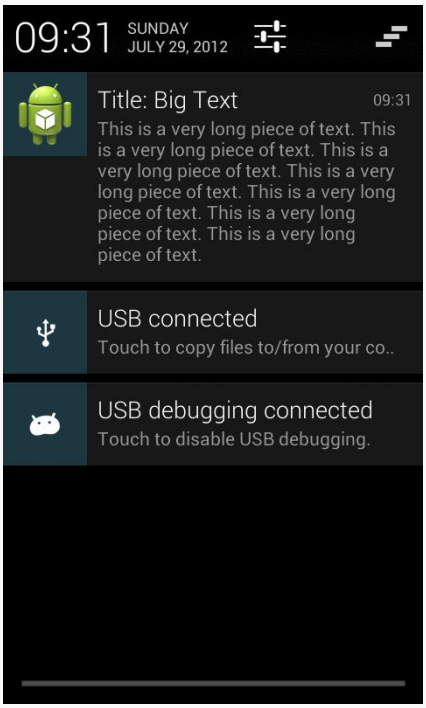


Figure 296: BigText Notification

We also have the Inbox style, which is the same basic concept but designed, for several discrete lines of text:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

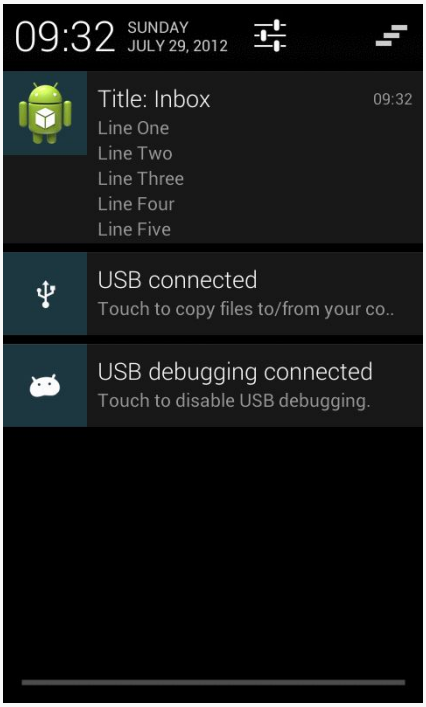


Figure 297: Inbox Notification

And, we have the `BigPicture` style, ideal for a photo, album cover, or the like:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

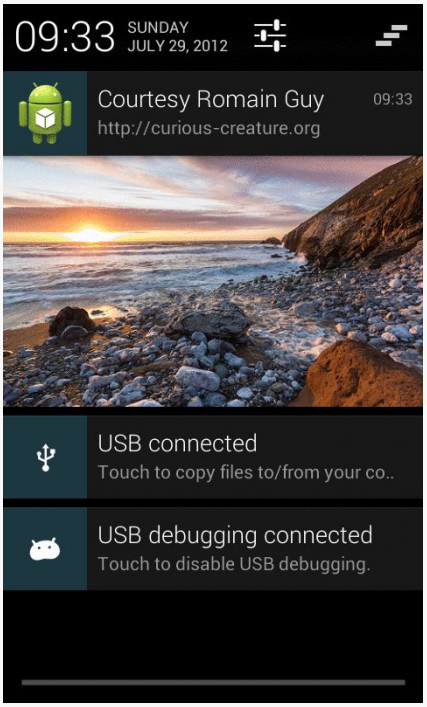


Figure 298: BigPicture Notification

(as noted in the screenshot, the photo is courtesy of Romain Guy, a former engineer on the core Android team and photography buff)

The Builders

`Notification.Builder` and `NotificationCompat.Builder` have been enhanced to support these new styles. Specifically:

- There is an `addAction()` method on the `Builder` class to define the action buttons, in terms of icon, caption, and `PendingIntent` that should be executed when the button is clicked
- There are style-specific builders, such as `Notification.InboxStyle`, that take a `Notification.Builder` and define the alternative expanded definition to be used when the `Notification` is at the top

The Sample

To see expanded notifications, take a peek at the `Notifications/BigNotify` sample application. This application consists of a single activity (`MainActivity`) that will raise a `Notification` and `finish()`, using `@style/Theme.NoDisplay` to suppress the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

activity's own UI. Hence, the result of running the app is to display the `Notification` and do nothing else. While silly, it minimizes the amount of ancillary code involved in the project.

The process of displaying an expanded `Notification` is to first create the basic `Notification`, containing what you want to display for any non-expanded circumstance:

- Older devices that cannot display expanded `Notifications`, or
- Newer devices where the `Notification` is not the top-most entry in the notification drawer, and therefore appears in the classic non-expanded form

Hence, in `onCreate()`, after getting our hands on a `NotificationManager`, we use `NotificationCompat.Builder` to create a regular `Notification`, wrapped in a private `buildNormal()` method:

```
  private NotificationCompat.Builder buildNormal() {
    NotificationCompat.Builder b=new NotificationCompat.Builder(this);

    b.setAutoCancel(true)
     .setDefaults(Notification.DEFAULT_ALL)
     .setContentTitle(getString(R.string.download_complete))
     .setContentText(getString(R.string.fun))
     .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
     .setSmallIcon(android.R.drawable.stat_sys_download_done)
     .setTicker(getString(R.string.download_complete))
     .setPriority(Notification.PRIORITY_HIGH)
     .addAction(android.R.drawable.ic_media_play,
                getString(R.string.play),
                buildPendingIntent(Settings.ACTION_SETTINGS));

    return(b);
  }
```

Most of what `buildNormal()` does is the same sort of stuff we saw with `NotificationCompat.Builder` earlier in this chapter. There are two things, though, that are new:

1. We call `setPriority()` to set the priority of the `Notification` to `PRIORITY_HIGH`. This means that this `Notification` *may* be displayed higher in the notification drawer than it might ordinarily appear.
2. We call `addAction()` to add an action button to the `Notification`, to be shown in the expanded form. We are able to supply an icon, caption, and `PendingIntent`, the latter created by a `buildPendingIntent()` method that

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

wraps our desired `Intent` action string (here, `Settings.ACTION_SETTINGS`) in an `Intent`:

```
private PendingIntent buildPendingIntent(String action) {
  Intent i=new Intent(action);

  return(PendingIntent.getActivity(this, 0, i, 0));
}
```

Ordinarily, we might use this `Builder` directly, to raise the `Notification` we described. And, if we just wanted the action button to appear and nothing else new in the expanded form, we could do just that. But in our case, we also want to change the look of the expanded widget to a new style, `InboxStyle`. To do that, we need to wrap our `Builder` in a `NotificationCompat.InboxStyle` builder:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);
  NotificationCompat.Builder normal=buildNormal();
  NotificationCompat.InboxStyle big=
      new NotificationCompat.InboxStyle(normal);

  mgr.notify(NOTIFY_ID,
             big.setSummaryText(getString(R.string.summary))
                .addLine(getString(R.string.entry))
                .addLine(getString(R.string.another_entry))
                .addLine(getString(R.string.third_entry))
                .addLine(getString(R.string.yet_another_entry))
                .addLine(getString(R.string.low)).build());

  finish();
}
```

Each of these "big" builders has a set of methods that are unique to that type of builder to configure the look beyond what a standard `Notification` might have. Specifically, in this case, we call:

- `setSummaryText()`, to provide "the first line of text after the detail section in the big form of the template", in the words of the JavaDocs, though this does not necessarily mean what you think it does
- `addLine()`, to append several lines of text to appear in the `Notification`

It is the `Notification` created by our `NotificationCompat.InboxStyle` builder that we use with the call to `notify()` on `NotificationManager`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Results

If we run our app, we get this:

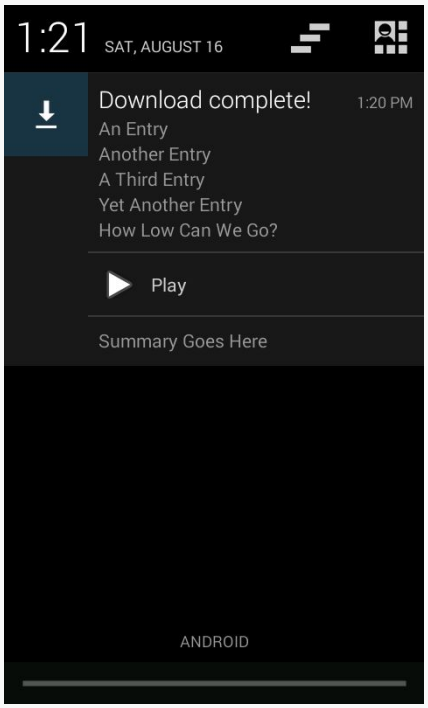


Figure 299: Expanded Notification in Drawer, on Android 4.4

From top to bottom, we have:

- Our content text
- Our appended lines of text
- Our action button
- Our summary text

Note that this is the appearance when we are in expanded mode, at the top of the notification drawer. If our `Notification` is not at the top, or if it is displayed on a pre–4.1 device, the appearance is the normal style, as defined by our `buildNormal()` method, though on Android 4.1+ devices the user can use a two-finger downward swipe gesture to expand the un-expanded `Notification`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Target Requirement

Note that to use action buttons successfully, you need to have your `android:targetSdkVersion` set to 11 or higher. Technically, they will work with lower values, but the contents of the button will be rendered incorrectly, with a gray-on-gray color scheme that makes the buttons all but unreadable. Using 11 or higher will cause the buttons to be rendered with an appropriate color scheme.

Foreground Services

If you have a service that will run for a substantial period of time, there is a risk that your process will still be terminated. That could be triggered by the user, or it could be the OS's own decision, based on the age of your process.

Generally speaking, this is a good thing for the user, because too many developers "leak" services, causing them to run unnecessarily, without adding value to the user, and tying up system RAM as a result.

But, what about services that *are* delivering value to the user for a long period? For example, what about a music player, where, in theory, the service is delivering value until the user presses some sort of "stop" button somewhere to turn off the music?

For those sorts of situations, you can flag a service as being a "foreground service".

Isn't "Foreground Service" an Oxymoron?

You might be forgiven for thinking that "foreground" and "service" are not designed to go together.

Partly, that is because we have overloaded the term "foreground".

A foreground service is *not* one that somehow takes over the screen. A foreground service *is* one that runs with foreground priority. That means:

- It will be treated similarly to the app that is in the UI foreground, from the standpoint of determining processes eligible for termination
- It will be classified as foreground from a CPU standpoint, rather than being relegated to the standard background process group

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The former is what many developers want: a service (and process) that will not go away.

The latter is what many users fear: a service (and process) that is capable of stealing chunks of CPU time away from the game, video, or whatever else is truly in the foreground from a UI standpoint.

Services themselves, while useful, are best when used *sparingly*, only running when they are actively delivering value to the user. "This goes double" for foreground services.

Putting Your Service in the Foreground

Putting a service into the foreground is a matter of calling `startForeground()`. This method takes two parameters, the same two parameters that you would pass to `notify()` of `NotificationManager`:

- A prepared `Notification`
- A unique ID for that `Notification`

Android will then display the `Notification`. So long as the `Notification` is visible, your app's process will be given foreground priority.

You undo this by calling `stopForeground()`. `stopForeground()` takes a boolean parameter, indicating if the `Notification` should be removed (`true`) or not (`false`). Typically, you will pass `true`, so the `Notification` only clutters up the screen while you need it.

The `Notifications/Foreground` sample project is a clone of the `Notifications/DownloadNotify` sample that opened this chapter, adding in the use of `startForeground()` and `stopForeground()`.

Towards the top of `onHandleIntent()`, we call `startForeground()`, to *really* ensure that our process will remain intact long enough to complete the requested download:

```
      startForeground(FOREGROUND_ID,
                      buildForegroundNotification(filename));
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This, in turn, uses a `buildForegroundNotification()` method to build the `Notification` that will be displayed while the service is categorized as being in the foreground:

```
  private Notification buildForegroundNotification(String filename) {
    NotificationCompat.Builder b=new NotificationCompat.Builder(this);

    b.setOngoing(true);

    b.setContentTitle(getString(R.string.downloading))
     .setContentText(filename)
     .setSmallIcon(android.R.drawable.stat_sys_download)
     .setTicker(getString(R.string.downloading));

    return(b.build());
  }
```

Note that we use `setOngoing(true)`, to indicate that this is an "ongoing" operation. This precludes the user from removing the `Notification` manually, as doing that would drop our process out of foreground priority.

Towards the end of `onHandleIntent()`, we call `stopForeground()`, before calling `raiseNotification()`:

```
      stopForeground(true);
      raiseNotification(i, output, null);
```

There is a similar `stopForeground()` call in the `catch` block that raises the failure `Notification` in case of an I/O error.

In both cases, we pass `true` to `stopForeground()` to remove the `Notification`. From the user's perspective, we could just as easily have passed `false`, as the `Notification` used with `startForeground()` will also be removed once our service is destroyed, which will happen shortly after `onHandleIntent()` ends.

The Malformed Notification

Of course, some developers do not play nicely with the other kids.

A technique that had been around for a while was for an app to pass an intentionally-flawed `Notification` to `startForeground()`. While Android would blow up silently somewhere internally actually trying to display the `Notification`, the foreground status was still granted. This resulted in behavior reminiscent of the long-since-deprecated `setForeground()` method.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`setForeground()` allowed a service to get foreground priority with no repercussions. Not surprisingly, lots of developers used it, as they decided that their app was more important than any other apps on the device. `setForeground()` was replaced by `startForeground()`, adding in the `Notification` requirement, to put a "cost" on foreground status. The malformed-`Notification` trick allowed developers to avoid that cost.

In Android 4.3, if you pass a malformed `Notification` to `startForeground()`, Android will create one for you, featuring your app's launcher icon, and use it instead. Hence, on Android 4.3 and higher, you cannot hide your foreground status from the user.

Needless to say, when this behavior was mentioned in a Google+ post by Dianne Hackborn, a number of developers complained.

That being said, if you were using this hack, you will really need to consider alternative strategies, as users do get irritated with everlasting services when they are made aware of them:

- If you were using a foreground service to fix some bug in your app caused by your process being terminated and later restarted (e.g., `START_STICKY`), fix the bug.
- Consider making foreground behavior optional. Since the decision of marking a service as a foreground service is made in Java, not the manifest, it is relatively easy for you to add a checkbox to your app settings to allow the user to indicate whether the benefits of foreground-ness are worth the cost of having the `Notification` around.
- Consider making the service optional, if the foreground status is needed for the service to be useful, but the features enabled by the service itself are not absolutely essential.
- There is no need to have more than one simultaneous foreground service. One service can have multiple threads to do disparate operations.

Disabled Notifications

Because apps have the ability to display larger-than-normal `Notifications`, plus force them towards the top of the list via priority levels, Android has given users the ability to disable `Notifications` on a per-app basis. The degree of control, and the way the user sets up that control, depends upon Android version.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android 4.x

Users visiting an app's page in Settings will see a "Show notifications" checkbox:

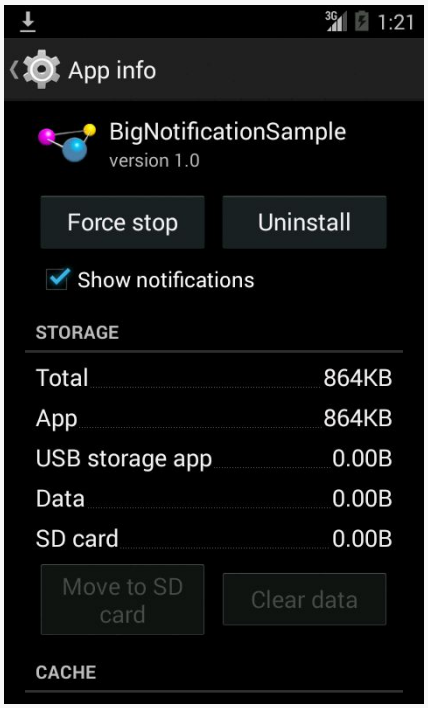


Figure 300: Show Notifications Checkbox, on Android 4.4

If the user unchecks the checkbox and agrees on the resulting confirmation dialog, your requests to raise a `Notification` will be largely ignored. An error message will appear in LogCat ("Suppressing notification from package ... by user request"), but no exception will be raised. Further, there does not appear to be an API for you to determine if the notification will actually be displayed.

Also note that, on Android 4.2+, if the user blocks notifications, it also blocks `Toast` requests from your app.

And, also note that this setting survives an uninstall of your app. If the user unchecks this checkbox, uninstalls your app, then reinstalls your app, the checkbox is still unchecked, meaning that notifications will still be blocked.

The one notable exception to this blocking, as of Android 4.3, is that the `Notification` associated with a foreground service will not be blocked. It will always appear, even if the user unchecked "Show notifications" for your app in Settings.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android 5.0+

In the "Sound & notification" area of Settings, the user can tap on an "App notifications" option, and from there choose an app. This brings up a screen where the user can "block" (i.e., disable) notifications:

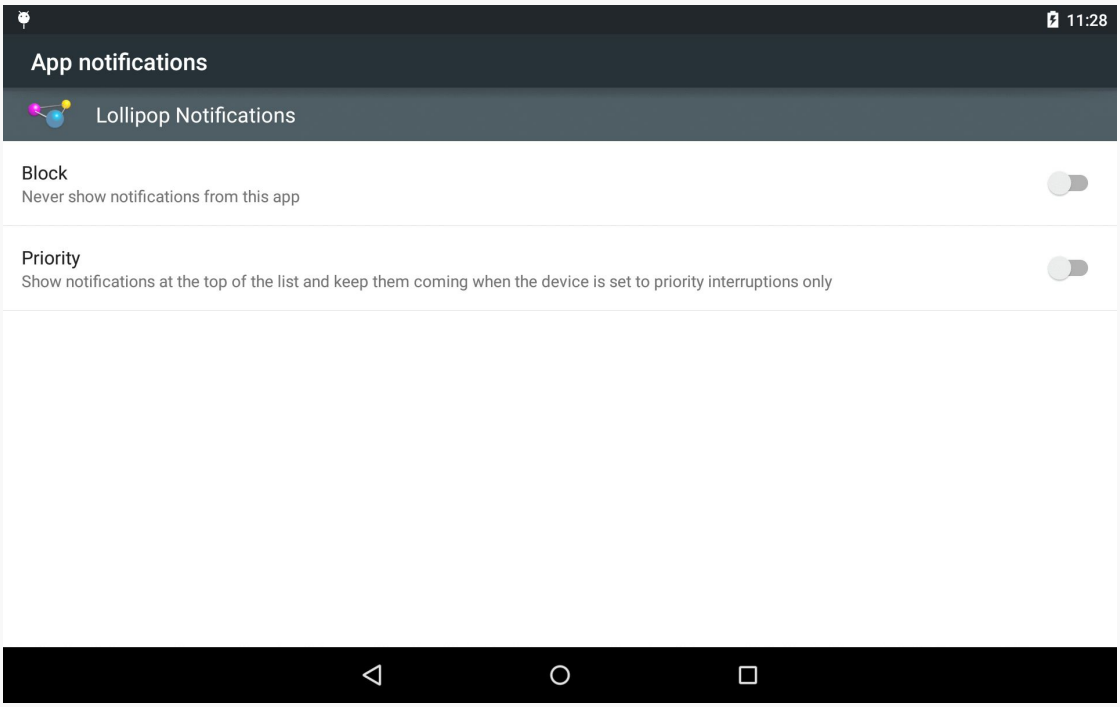


Figure 301: "App notifications" in Settings, on Android 5.0

The top "Block" `SwitchPreference`, if toggled on, will prevent app notifications from being displayed.

The bottom "Priority" `SwitchPreference`, if toggled on, marks this app's notifications as being "priority". Then, in the main "Sound & notifications" area of Settings, the user can tap on an "Interruptions" option:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

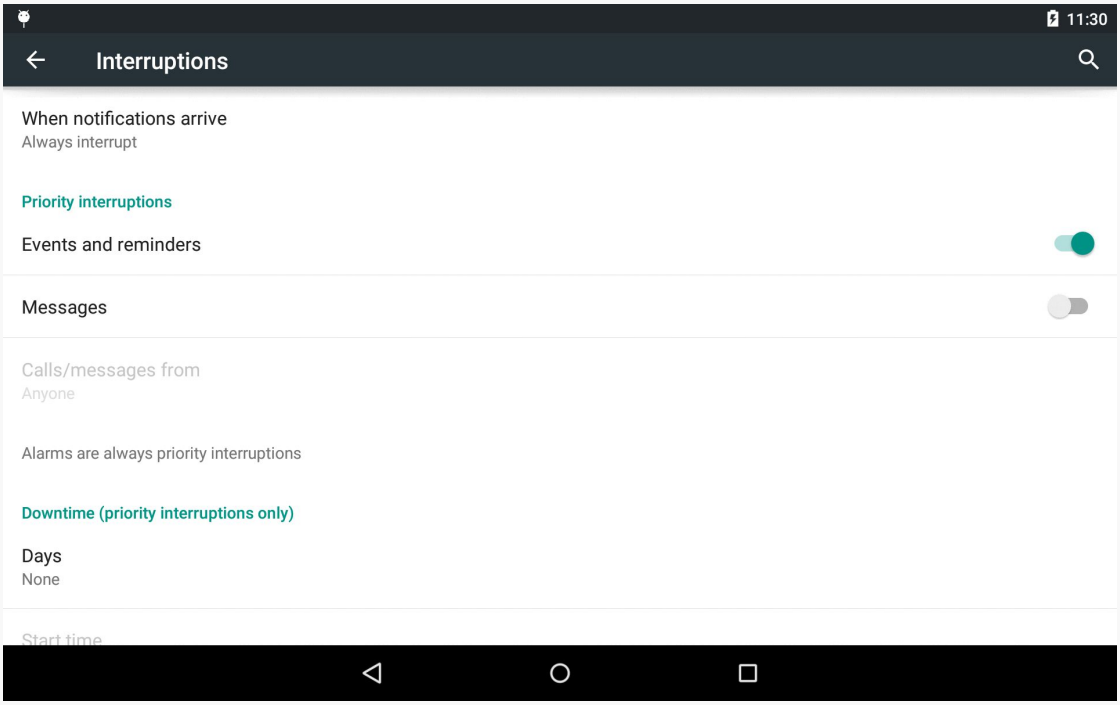


Figure 302: "Interruptions" in Settings, on Android 5.0

If the user toggles the "When notifications arrive" option to "Allow only priority interruptions", then those apps that the user configures as "Priority" in "App notifications" will behave normally. Other apps' notifications will appear, but will not play a ringtone or vibrate the device.

If the user toggles the "When notifications arrive" option to "Don't interrupt", all notifications — even those marked as "Priority" — will have their ringtones and vibrations suppressed.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #18 - Notifying the User

In the last tutorial, we added automatic updating. However, the user will not know that the book was updated in the background, unless they open the book and see an update. It would be nice to let the user know that an update succeeded, if `EmPubLiteActivity` is not in the foreground, and a `Notification` is a likely solution.

But we only want to show the `Notification` if our UI is not in the foreground, as it would be silly to show a `Notification` telling the user to go view the book that the user is already viewing. Hence, we will want to adjust our use of greenrobot's EventBus to allow us to update the UI if it is around, or else show the `Notification`.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Raising the Notification

We need to arrange to find out when the `BookUpdatedEvent` is not handled by the `ModelFragment` and raise a `Notification` in that case.

First, create two new string resources:

- `R.string.update_desc`, with a value of `Click here to open the updated book!`
- `R.string.update_complete`, with a value of `EmPub Lite Updated!`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Next, add a `NOTIFY_ID` static data member to `DownloadCheckService`:

```
  private static final int NOTIFY_ID=1337;
```

This will be used as a unique identifier of our notification when we raise it.

Then, modify `doWakefulWork()` in `DownloadCheckService` to handle the events a bit differently, registering and unregistering from the bus before posting:

```
  @Override
  protected void doWakefulWork(Intent intent) {
    try {
      String url=getUpdateUrl();

      if (url != null) {
        File book=download(url);
        File updateDir=new File(getFilesDir(), UPDATE_BASEDIR);

        updateDir.mkdirs();
        unzip(book, updateDir);
        book.delete();

        EventBus.getDefault().register(this);
        EventBus.getDefault().post(new BookUpdatedEvent());
        EventBus.getDefault().unregister(this);
      }
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(),
          "Exception downloading update", e);
    }
  }
```

Note that `post()`, as we are now using it, will deliver its event to `ModelFragment` (if it is registered) on the same thread we are using to call `post()`, and so this work will happen synchronously.

Finally, add an `onEvent()` method to `DownloadCheckService`:

```
  public void onEvent(NoSubscriberEvent event) {
    NotificationCompat.Builder builder=new NotificationCompat.Builder(this);
    Intent toLaunch=new Intent(this, EmPubLiteActivity.class);
    PendingIntent pi=PendingIntent.getActivity(this, 0, toLaunch, 0);

    builder.setAutoCancel(true).setContentIntent(pi)
        .setContentTitle(getString(R.string.update_complete))
        .setContentText(getString(R.string.update_desc))
        .setSmallIcon(android.R.drawable.stat_sys_download_done)
        .setTicker(getString(R.string.update_complete));
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    NotificationManager mgr=
        ((NotificationManager)getSystemService(Context.NOTIFICATION_SERVICE));

    mgr.notify(NOTIFY_ID, builder.build());
  }
```

Here, we:

- Craft an `Intent` identifying `EmPubLiteActivity`
- Wrap it in a `getActivity()` `PendingIntent`
- Wrap that in a `Notification` created using a `Notification.Builder`, configured to let the user know that a book update is ready for download
- Raise that `Notification` using a `NotificationManager`

Because we are listening for `NoSubscriberEvent`, this will be triggered when our `post()` of the `BookUpdatedEvent` fails to be picked up by any subscriber. That means that our UI did not update itself to reflect the book update, and therefore it is the service's job to let the user know about the update.

To test this, repeat the test from Step #5 of the previous tutorial. You should see the `Notification` appear once the update has completed. Sliding open the notification drawer and tapping on the notification should bring up the book for reading.

In Our Next Episode…

… we will move some fragments into a sidebar on large-screen devices, like tablets.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Large-Screen Strategies and Tactics

So far, we have been generally ignoring screen size. With the vast majority of Android devices being in a fairly narrow range of sizes (3" to just under 5"), ignoring size while learning is not a bad approach. However, when it comes time to create a production app, you are going to want to strongly consider how you are going to handle other sizes, mostly larger ones (e.g., tablets).

Objective: Maximum Gain, Minimum Pain

What you want is to be able to provide a high-quality user experience without breaking your development budget — time *and* money — in the process.

An app designed around a phone, by default, may look fairly lousy on a tablet. That is because Android is simply going to try to stretch your layouts and such to fill the available space. While that will work, technically, the results may be unpleasant, or at least ineffective. If we have the additional room, it would be nice to allow the user to do something with that room.

At the same time, though, you do not have an infinite amount of time to be dealing with all of this. After all, there are a variety of tablet sizes. While ~7" and ~10" screens are the most common, there are certainly others that are reasonably popular (e.g., Amazon's Kindle Fire HD 8.9").

The Fragment Strategy

Some apps will use the additional space of a large screen directly. For example, a painting app would use that space mostly to provide a larger drawing canvas upon which the user can attempt to become the next Rembrandt, Picasso, or Pollock. The

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

app might elect to make more tools available directly on the screen as well, versus requiring some sort of pop-up to appear to allow the user to change brush styles, choose a different color, and so forth.

However, this can be a lot of work.

Some apps can make a simplifying assumption: the tablet UI is really a bunch of phone-sized layouts, stitched together. For example, if you take a 10" tablet in landscape, it is about the same size as two or three phones side-by-side. Hence, one could imagine taking the smarts out of a few activities and having them be adjacent to one another on a tablet, versus having to be visible only one at a time as they are on phones.

For example, consider Gmail.

On a phone, you see conversations in a particular label on one screen:

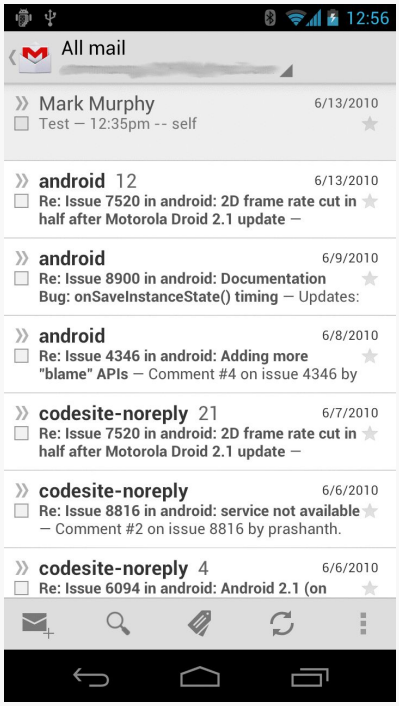


Figure 303: Gmail, On a Galaxy Nexus, Showing Conversations

... and the list of labels on another screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

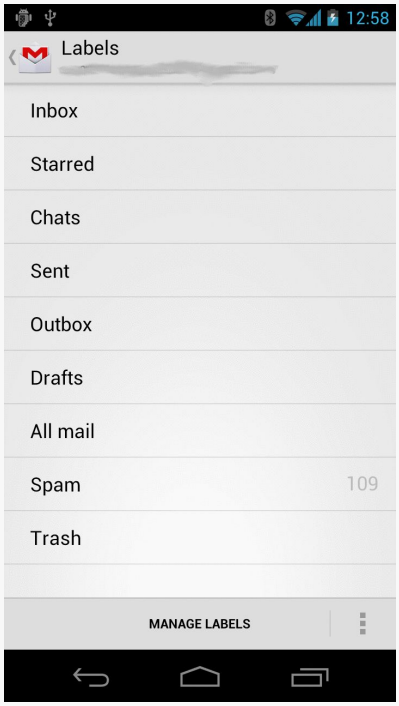


Figure 304: Gmail, On a Galaxy Nexus, Showing Labels

... and the list of messages in some selected conversation in a third screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

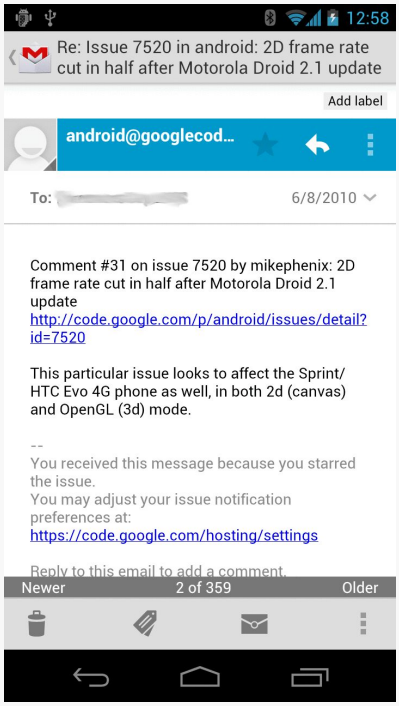


Figure 305: Gmail, On a Galaxy Nexus, Showing Messages

Whereas on a 7" tablet, you see the list of labels and the conversations in a selected label at the same time:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

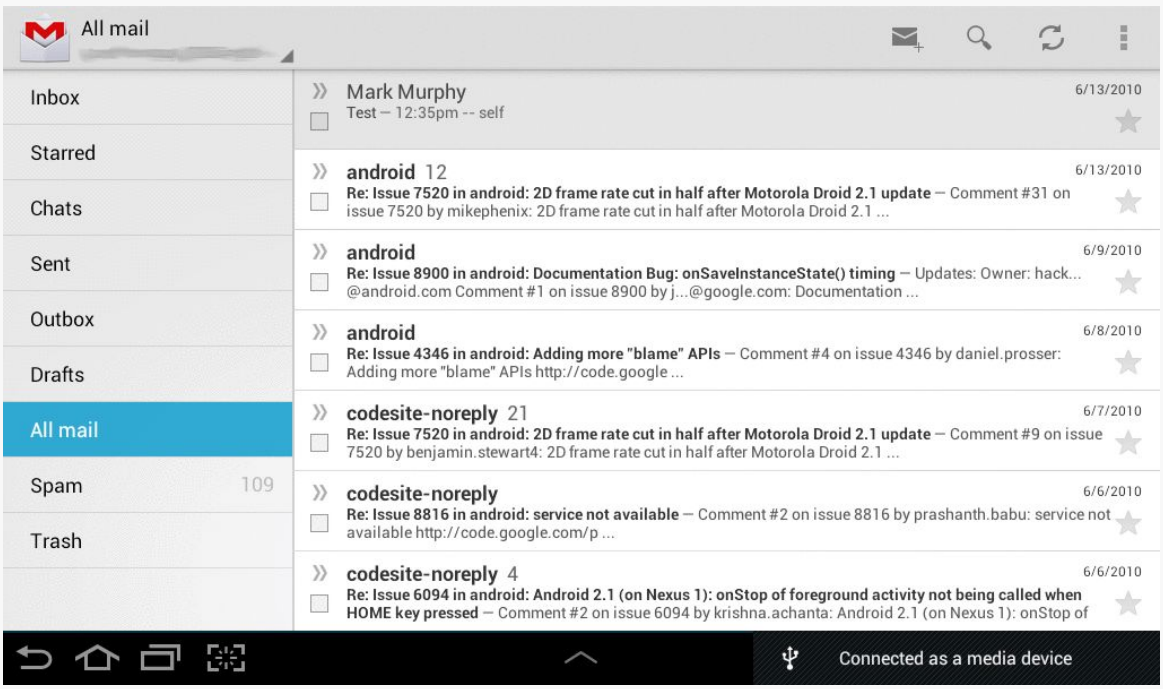


Figure 306: Gmail, On a Galaxy Tab 2, Showing Labels and Conversations

On that 7" tablet, tapping on a specific conversation brings up the list of messages for that conversation in a new screen. But, on a 10" tablet, tapping on a specific conversation shows it, plus the list of conversations, side-by-side:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

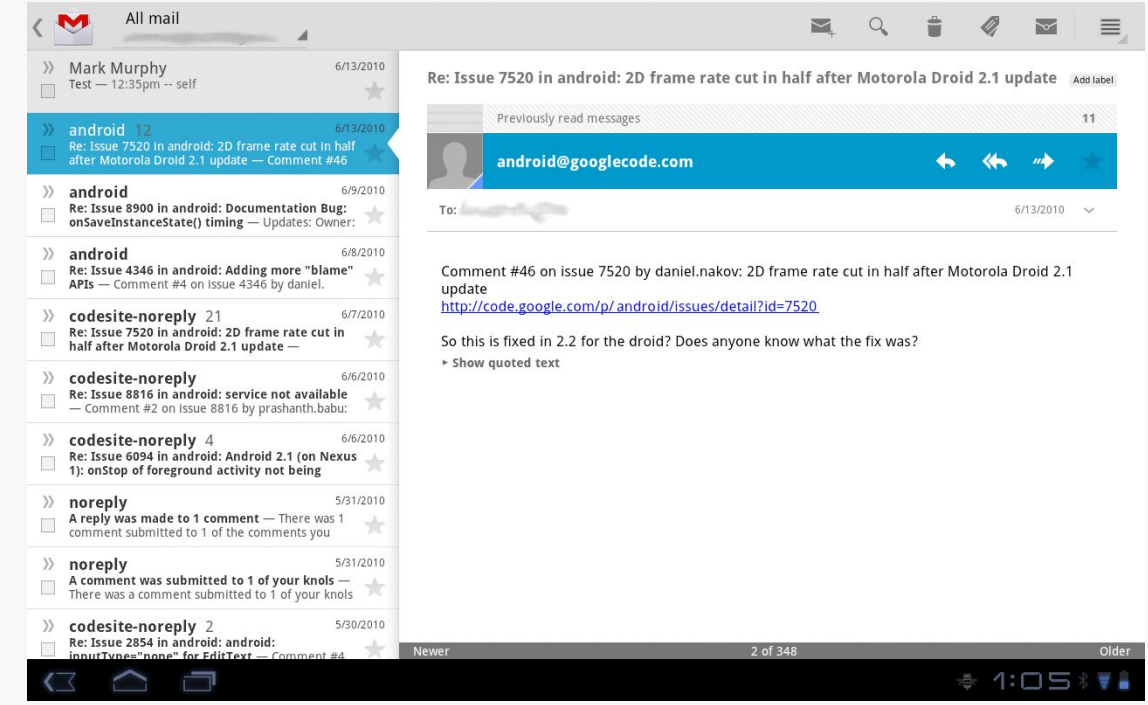


Figure 307: Gmail, On a XOOM, Showing Conversations and Messages

Yet all of that was done with one app with very little redundant logic, by means of fragments.

The list-of-labels, list-of-conversations, and list-of-messages bits of the UI were implemented as fragments. On a smaller screen (e.g., a phone), each one is displayed by an individual activity. Yet, on a larger screen (e.g., a tablet), more than one fragment is displayed by a single activity. In fact — though it will not be apparent from the static screenshots — on the 10" tablet, the activity showed *all three fragments*, using animated effects to slide the list of labels off-screen and the list of conversations over to the left slot when the user taps on a conversation to show the messages.

The vision, therefore, is to organize your UI into fragments, then choose which fragments to show in which circumstances based on available screen space:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

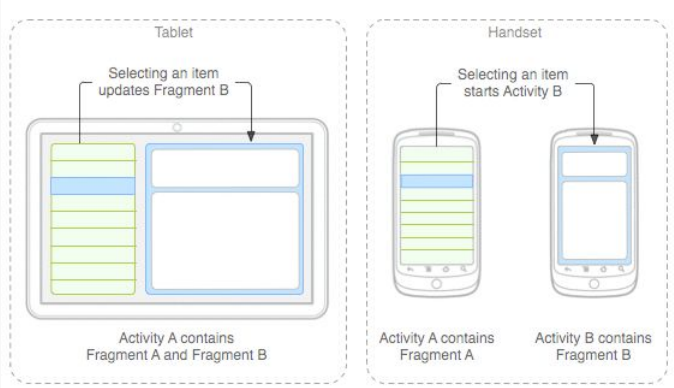


Figure 308: Tablets vs. Handsets (image courtesy of Android Open Source Project)

Changing Layout

One solution is to say that you have the same fragments for all devices and all configurations, but that the sizing and positioning of those fragments varies. This is accomplished by using different layouts for the activity, ones that provide the sizing and positioning rules for the fragments.

So far, most of our fragment examples have been focused on activities with a single fragment, like you might use on smaller screens (e.g., phones). However, activities can most certainly have more than one fragment, though you will need to provide the "slots" into which to plug those fragments.

For example, you could have the following in `res/layout-w720dp/main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <FrameLayout
    android:id="@+id/countries"
    android:layout_weight="30"
    android:layout_width="0px"
    android:layout_height="match_parent"
  />
  <FrameLayout
    android:id="@+id/details"
    android:layout_weight="70"
    android:layout_width="0px"
    android:layout_height="match_parent"
  />
</LinearLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here we have a horizontal `LinearLayout` holding a pair of `FrameLayout` containers. Each of those `FrameLayout` containers will be a slot to load in a fragment, using code like:

```
getSupportFragmentManager().beginTransaction()
                                   .add(R.id.countries, someFragmentHere)
                                   .commit();
```

In principle, you could also have a `res/layout-h720dp/main.xml` that holds both of the same `FrameLayout` containers, but just in a `vertical` `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <FrameLayout
    android:id="@+id/countries"
    android:layout_weight="30"
    android:layout_width="0px"
    android:layout_height="match_parent"
  />
  <FrameLayout
    android:id="@+id/details"
    android:layout_weight="70"
    android:layout_width="0px"
    android:layout_height="match_parent"
  />
</LinearLayout>
```

As the user rotates the device, the fragments will go in their appropriate slots.

Changing Fragment Mix

However, for larger changes in screen size, you will probably need to have larger changes in your fragments. The most common pattern is to have fewer fragments on-screen for an activity on a smaller-screen device (e.g., one fragment at a time on a phone) and more fragments on-screen for an activity on a larger-screen device (e.g., two fragments at a time on a tablet).

So, for example, as the counterpart to the `res/layout-w720dp/main.xml` shown in the previous section, you might have a `res/layout/main.xml` that looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/countries"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:layout_width="match_parent"
  android:layout_height="match_parent"
/>
```

This provides a single slot, `R.id.countries`, for a fragment, one that fills the screen. For a larger-screen device, held in landscape, you would use the two-fragment layout; for anything else (e.g., tablet in portrait, or phone in any orientation), you would use the one-fragment layout.

Of course, the content that belongs in the second fragment would have to show up *somewhere*.

Sometimes, when you add another fragment for a large screen, you only want it to be there some of the time. For example, a digital book reader (like the one we are building in the tutorials) might normally take up the full screen with the reading fragment, but might display a sidebar fragment based upon an action bar item click or the like. If you would like the BACK button to reverse your `FragmentTransaction` that added the second fragment — so pressing BACK removes that fragment and returns you to the single-fragment setup — you can add `addToBackStack()` as part of your `FragmentTransaction` construction:

```
getSupportFragmentManager().beginTransaction()
                           .addToBackStack(null)
                           .replace(R.id.sidebar, f)
                           .commit();
```

We will see this in the next tutorial.

The Role of the Activity

So, what is the activity doing?

First, the activity is the one loading the overall layout, the one indicating which fragments should be loaded (e.g., the samples shown above). The activity is responsible for populating those "slots" with the appropriate fragments. It can determine which fragments to create based on which slots exist, so it would only try to create a fragment to go in `R.id.details` if there actually is an `R.id.details` slot to use.

Next, the activity is responsible for handling any events that are triggered by UI work in a fragment (e.g., user clicking on a `ListView` item), whose results should impact other fragments (e.g., displaying details of the clicked-upon `ListView` item). The activity knows which fragments exist at the present time. So, the activity can

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

either call some method on the second fragment if it exists, or it can call `startActivity()` to pass control to another activity that will be responsible for the second fragment if it does not exist in the current activity.

Finally, the activity is generally responsible for any model data that spans multiple fragments. Whether that model data is held in a "model fragment" (as outlined in the chapter on fragments) or somewhere else is up to you.

Fragment Example: The List-and-Detail Pattern

This will make a bit more sense as we work through another example, this time focused on a common pattern: a list of something, where clicking on the list brings up details on the item that was clicked upon. On a larger-screen device, in landscape, both pieces are typically displayed at the same time, side-by-side. On smaller-screen devices, and sometimes even on larger-screen devices in portrait, only the list is initially visible — tapping on a list item brings up some other activity to display the details.

Describing the App

The sample app for this section is `LargeScreen/EU4You`. This app has a list of member nations of the European Union (EU). Tapping on a member nation will display the mobile Wikipedia page for that nation in a `WebView` widget.

The data model — such as it is and what there is of it — consists of a `Country` class which holds onto the country name (as a string resource ID), flag (as a drawable resource ID), and mobile Wikipedia URL (as another string resource ID):

```
Country(int name, int flag, int url) {
  this.name=name;
  this.flag=flag;
  this.url=url;
}
```

The `Country` class has a static `ArrayList` of `Country` objects representing the whole of the EU, initialized in a static initialization block:

```
  static {
    EU.add(new Country(R.string.austria, R.drawable.austria,
                       R.string.austria_url));
    EU.add(new Country(R.string.belgium, R.drawable.belgium,
                       R.string.belgium_url));
    EU.add(new Country(R.string.bulgaria, R.drawable.bulgaria,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                    R.string.bulgaria_url));
    EU.add(new Country(R.string.cyprus, R.drawable.cyprus,
                    R.string.cyprus_url));
    EU.add(new Country(R.string.czech_republic,
                    R.drawable.czech_republic,
                    R.string.czech_republic_url));
    EU.add(new Country(R.string.denmark, R.drawable.denmark,
                    R.string.denmark_url));
    EU.add(new Country(R.string.estonia, R.drawable.estonia,
                    R.string.estonia_url));
    EU.add(new Country(R.string.finland, R.drawable.finland,
                    R.string.finland_url));
    EU.add(new Country(R.string.france, R.drawable.france,
                    R.string.france_url));
    EU.add(new Country(R.string.germany, R.drawable.germany,
                    R.string.germany_url));
    EU.add(new Country(R.string.greece, R.drawable.greece,
                    R.string.greece_url));
    EU.add(new Country(R.string.hungary, R.drawable.hungary,
                    R.string.hungary_url));
    EU.add(new Country(R.string.ireland, R.drawable.ireland,
                    R.string.ireland_url));
    EU.add(new Country(R.string.italy, R.drawable.italy,
                    R.string.italy_url));
    EU.add(new Country(R.string.latvia, R.drawable.latvia,
                    R.string.latvia_url));
    EU.add(new Country(R.string.lithuania, R.drawable.lithuania,
                    R.string.lithuania_url));
    EU.add(new Country(R.string.luxembourg, R.drawable.luxembourg,
                    R.string.luxembourg_url));
    EU.add(new Country(R.string.malta, R.drawable.malta,
                    R.string.malta_url));
    EU.add(new Country(R.string.netherlands, R.drawable.netherlands,
                    R.string.netherlands_url));
    EU.add(new Country(R.string.poland, R.drawable.poland,
                    R.string.poland_url));
    EU.add(new Country(R.string.portugal, R.drawable.portugal,
                    R.string.portugal_url));
    EU.add(new Country(R.string.romania, R.drawable.romania,
                    R.string.romania_url));
    EU.add(new Country(R.string.slovakia, R.drawable.slovakia,
                    R.string.slovakia_url));
    EU.add(new Country(R.string.slovenia, R.drawable.slovenia,
                    R.string.slovenia_url));
    EU.add(new Country(R.string.spain, R.drawable.spain,
                    R.string.spain_url));
    EU.add(new Country(R.string.sweden, R.drawable.sweden,
                    R.string.sweden_url));
    EU.add(new Country(R.string.united_kingdom,
                    R.drawable.united_kingdom,
                    R.string.united_kingdom_url));
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

CountriesFragment

The fragment responsible for rendering the list of EU nations is `CountriesFragment`. It is a `ListFragment`, using a `CountryAdapter` to populate the list:

```
class CountryAdapter extends ArrayAdapter<Country> {
  CountryAdapter() {
    super(getActivity(), R.layout.row, R.id.name, Country.EU);
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    CountryViewHolder wrapper=null;

    if (convertView == null) {
      convertView=
          LayoutInflater.from(getActivity()).inflate(R.layout.row,
                                                     parent, false);
      wrapper=new CountryViewHolder(convertView);
      convertView.setTag(wrapper);
    }
    else {
      wrapper=(CountryViewHolder)convertView.getTag();
    }

    wrapper.populateFrom(getItem(position));

    return(convertView);
  }
}
```

This adapter is somewhat more complex than the ones we showed in the chapter on selection widgets. We will get into what `CountryAdapter` is doing, and the `CountryViewHolder` it references, in a later chapter of this book. Suffice it to say for now that the rows in the list contain both the country name and its flag.

When the user taps on a row in our `ListView`, something needs to happen – specifically, the details of that country need to be displayed. However, displaying those details is not the responsibility of `CountriesFragment`, as it simply displays the list of countries and nothing else. Hence, we need to pass the event up to the hosting activity to handle.

We could accomplish this using an event bus, as seen in other examples earlier in the book. The `EU4You` series of samples, though, use a different approach, referred to as the contract pattern. In this pattern, the fragment defines an interface, which is the "contract" that all hosting activities of that fragment must implement. This requirement is enforced by the superclass, `ContractListFragment`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
/***
  Copyright (c) 2013 Jake Wharton
  Portions Copyright (c) 2013 CommonsWare, LLC

  Licensed under the Apache License, Version 2.0 (the "License"); you may not
  use this file except in compliance with the License. You may obtain a copy
  of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required
  by applicable law or agreed to in writing, software distributed under the
  License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS
  OF ANY KIND, either express or implied. See the License for the specific
  language governing permissions and limitations under the License.

  From _The Busy Coder's Guide to Android Development_
    http://commonsware.com/Android
 */

// derived from https://gist.github.com/JakeWharton/2621173

package com.commonsware.android.eu4you;

import android.app.Activity;
import android.app.ListFragment;

public class ContractListFragment<T> extends ListFragment {
  private T contract;

  @SuppressWarnings("unchecked")
  @Override
  public void onAttach(Activity activity) {
    super.onAttach(activity);

    try {
      contract=(T)activity;
    }
    catch (ClassCastException e) {
      throw new IllegalStateException(activity.getClass()
                                              .getSimpleName()
          + " does not implement contract interface for "
          + getClass().getSimpleName(), e);
    }
  }

  @Override
  public void onDetach() {
    super.onDetach();

    contract=null;
  }

  public final T getContract() {
    return(contract);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`onAttach()` is called when the fragment has been attached to an activity, whether that is from when the activity was initially created, after a configuration change, or whenever. In those cases, we cast the activity to be the contract interface (provided via the data type in the declaration), raising an exception if the cast fails. Subclasses can then access the contract object via the `getContract()` method.

`CountriesFragment` inherits from `ContractListFragment` and defines its contract. Hence, any activity that hosts a `CountriesFragment` is responsible for implementing this contract interface, so we can call `onCountrySelected()` when the user clicks on a row in the list:

```
@Override
public void onListItemClick(ListView l, View v, int position, long id) {
  if (getContract().isPersistentSelection()) {
    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
    l.setItemChecked(position, true);
  }
  else {
    getListView().setChoiceMode(ListView.CHOICE_MODE_NONE);
  }

  getContract().onCountrySelected(Country.EU.get(position));
}
```

`CountriesFragment` also has quite a bit of code dealing with clicked-upon rows being in an "activated" state. This provides visual context to the user and is often used in the list-and-details pattern. For example, in the tablet renditions of Gmail shown earlier in this chapter, you will notice that the list on the left (e.g., list of labels) has one row highlighted with a blue background. This is the "activated" row, and it indicates the context for the material in the adjacent fragment (e.g., list of conversations in the label). Managing this "activated" state is a bit beyond the scope of this section, however, so we will delay discussion of that topic to a later chapter in this book.

DetailsFragment

The details to be displayed come in the form of a URL to a mobile Wikipedia page for a country, designed to be displayed in a `WebView`. The EU4You sample app makes use of the same `WebViewFragment` that we saw earlier in this book, such as in the tutorials. `DetailsFragment` itself, therefore, simply needs to expose some method to allow a hosting activity to tell it what URL to display:

```
package com.commonsware.android.eu4you;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.webkit.WebViewFragment;

public class DetailsFragment extends WebViewFragment {
  public void loadUrl(String url) {
    getWebView().loadUrl(url);
  }
}
```

You will notice that this fragment is not retained via `setRetainInstance()`. That is because, as you will see, we will not always be displaying this fragment. Fragments that are displayed in some configurations (e.g., landscape) but not in others (e.g., portrait), where a device might change between those configurations at runtime, cannot be retained without causing crashes.

The Activities

Our launcher activity is also named `EU4You`. It uses two of the layouts shown above. Both are `main.xml`, but one is in `res/layout-w720dp/`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <FrameLayout
    android:id="@+id/countries"
    android:layout_weight="30"
    android:layout_width="0px"
    android:layout_height="match_parent"
  />
  <FrameLayout
    android:id="@+id/details"
    android:layout_weight="70"
    android:layout_width="0px"
    android:layout_height="match_parent"
  />
</LinearLayout>
```

The other is in `res/layout/`:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/countries"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Both have a `FrameLayout` for the `CountriesFragment` (`R.id.countries`), but only the `res/layout-w720dp/` edition has a `FrameLayout` for the `DetailsFragment` (`R.id.details`).

Here is the complete implementation of the `EU4You` activity:

```
package com.commonsware.android.eu4you;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

public class EU4You extends Activity implements
    CountriesFragment.Contract {
  private CountriesFragment countries=null;
  private DetailsFragment details=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    countries=
        (CountriesFragment)getFragmentManager().findFragmentById(R.id.countries);

    if (countries == null) {
      countries=new CountriesFragment();
      getFragmentManager().beginTransaction()
                                .add(R.id.countries, countries)
                                .commit();
    }

    details=
        (DetailsFragment)getFragmentManager().findFragmentById(R.id.details);

    if (details == null && findViewById(R.id.details) != null) {
      details=new DetailsFragment();
      getFragmentManager().beginTransaction()
                                .add(R.id.details, details).commit();
    }
  }

  @Override
  public void onCountrySelected(Country c) {
    String url=getString(c.url);

    if (details != null && details.isVisible()) {
      details.loadUrl(url);
    }
    else {
      Intent i=new Intent(this, DetailsActivity.class);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      i.putExtra(DetailsActivity.EXTRA_URL, url);
      startActivity(i);
    }
  }

  @Override
  public boolean isPersistentSelection() {
    return(details != null && details.isVisible());
  }
}
```

The job of `onCreate()` is to set up the UI. So, we:

- See if we already have an instance of `CountriesFragment`, by asking our `FragmentManager` to give us the fragment in the `R.id.countries` slot — this might occur if we underwent a configuration change, as `CountriesFragment` would be recreated in that case
- If we do not have a `CountriesFragment` instance, create one and execute a `FragmentTransaction` to load it into `R.id.countries` of our layout
- Find the `DetailsFragment` (which, since `DetailsFragment` is not retained, should always return `null`, but, as they say, "better safe than sorry")
- If we do not have a `DetailsFragment` *and* the layout has a `R.id.details` slot, create a `DetailsFragment` and execute the `FragmentTransaction` to put it in that slot... but otherwise do nothing

The net result is that `EU4You` can correctly handle either situation, where we have both fragments or just one.

Similarly, the `onCountrySelected()` method (required by the `Contract` interface) will see if we have our `DetailsFragment` or not (and whether it is visible, or is hidden because we created it but it is not visible in the current screen orientation). If we do, we just call `loadUrl()` on it, to populate the `WebView`. If we do not have a visible `DetailsFragment`, we need to do something to display one. In principle, we could elect to execute a `FragmentTransaction` to replace the `CountriesFragment` with the `DetailsFragment`, but this can get complicated. Here, we start up a separate `DetailsActivity`, passing the URL for the chosen `Country` in an `Intent` extra.

`DetailsActivity` is similar:

```
package com.commonsware.android.eu4you;

import android.app.Activity;
import android.os.Bundle;

public class DetailsActivity extends Activity {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  public static final String EXTRA_URL=
      "com.commonsware.android.eu4you.EXTRA_URL";
  private String url=null;
  private DetailsFragment details=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    details=
        (DetailsFragment)getFragmentManager().findFragmentById(R.id.details);

    if (details == null) {
      details=new DetailsFragment();

      getFragmentManager().beginTransaction()
                                  .add(android.R.id.content, details)
                                  .commit();
    }

    url=getIntent().getStringExtra(EXTRA_URL);
  }

  @Override
  public void onResume() {
    super.onResume();

    details.loadUrl(url);
  }
}
```

We create the `DetailsFragment` and load it into the layout, capture the URL from the `Intent` extra, and call `loadUrl()` on the `DetailsFragment`. However, since we are executing a `FragmentTransaction`, the actual UI for the `DetailsFragment` is not created immediately, so we cannot call `loadUrl()` right away (otherwise, `DetailsFragment` will try to pass it to a non-existent `WebView`, and we crash). So, we delay calling `loadUrl()` to `onResume()`, at which point the `WebView` should exist.

The Results

On a larger-screen device, in landscape, we have both fragments, though there is nothing initially loaded into the `DetailsFragment`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

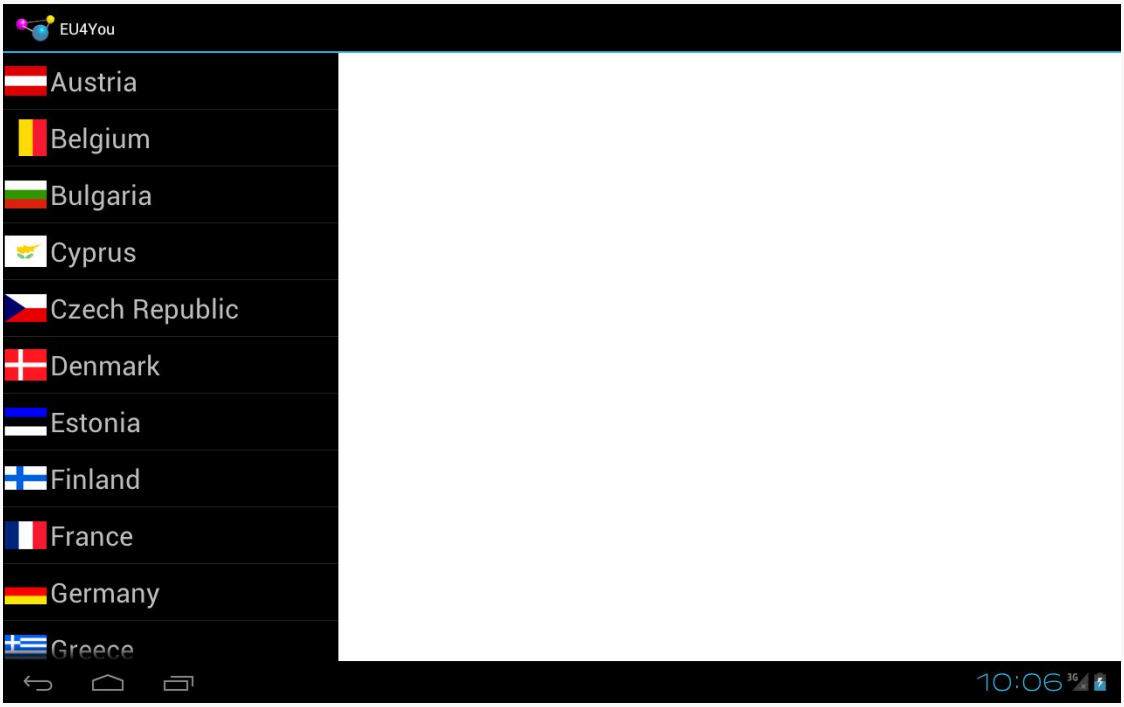


Figure 309: EU4You, On a Tablet Emulator, Landscape

Tapping on a country brings up the details on the right:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

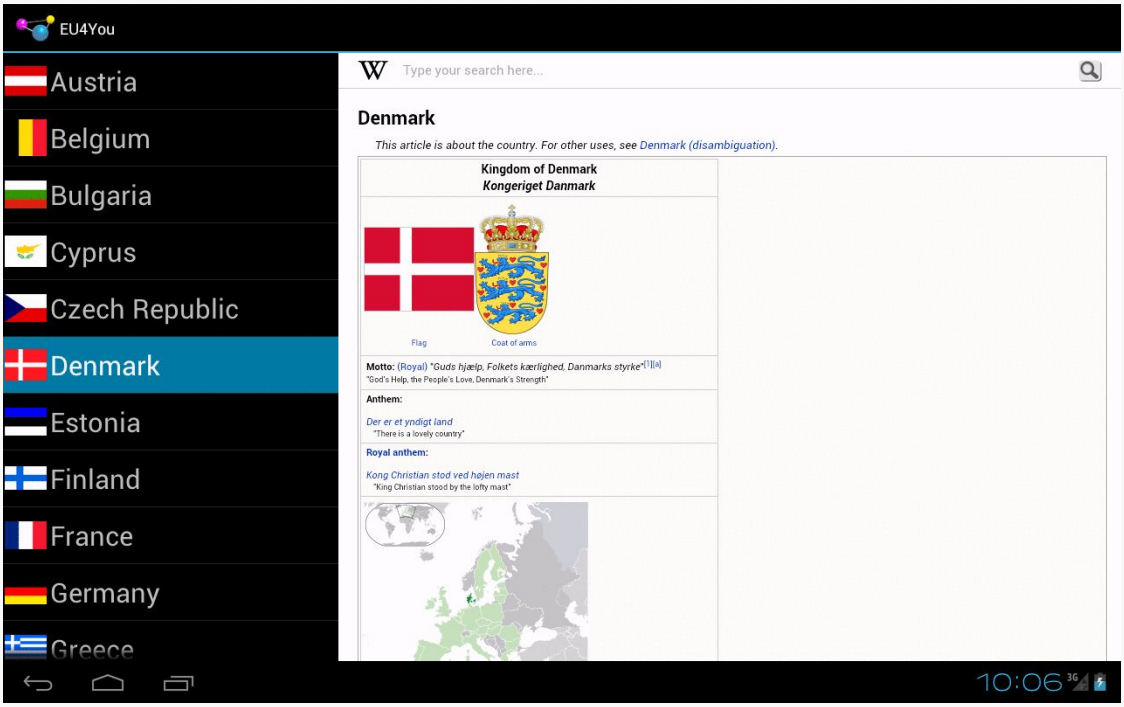


Figure 310: EU4You, On a Tablet Emulator, Landscape, With Details

In any other configuration, such as a smaller-screen device, we only see the `CountriesFragment` at the outset:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 311: EU4You, On a Phone Emulator

Tapping on a country brings up the `DetailsFragment` full-screen in the `DetailsActivity`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 312: EU4You, On a Phone Emulator, Showing Details

Other Master-Detail Strategies

The EU4You sample from above is one way of approaching this master-detail pattern. It is not the only one. In this section, will we review other implementations of EU4You that use other techniques for implementation the master-detail pattern.

Static CountriesFragment

In the original `EU4You` activity, both fragments were dynamic, each added via a `FragmentTransaction`. `DetailsFragment` has to be dynamic, as whether or not it is visible depends upon screen size and orientation. However, there is no particular need for our `CountriesFragment` to be dynamic, as you will see in the `LargeScreen/EU4YouStaticCountries` sample project.

Here, our single-pane layout uses a `<fragment>` element to wire in the `CountriesFragment`:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/countries"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:name="com.commonsware.android.eu4you3.CountriesFragment"
/>
```

Similarly, our dual-pane layout uses a `<fragment>` element for the `CountriesFragment`, alongside the `FrameLayout` for the details:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:baselineAligned="false"
  android:orientation="horizontal">

  <fragment
    android:id="@+id/countries"
    android:name="com.commonsware.android.eu4you3.CountriesFragment"
    android:layout_width="0px"
    android:layout_height="match_parent"
    android:layout_weight="30"/>

  <FrameLayout
    android:id="@+id/details"
    android:layout_width="0px"
    android:layout_height="match_parent"
    android:layout_weight="70"/>

</LinearLayout>
```

Our `onCreate()` for `EU4You` is simpler, in that we do not need to mess with the `CountriesFragment` at all:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    details=
        (DetailsFragment)getFragmentManager().findFragmentById(R.id.details);

    if (details == null && findViewById(R.id.details) != null) {
      details=new DetailsFragment();
      getFragmentManager().beginTransaction()
                          .add(R.id.details, details).commit();
    }
  }
```

Neither `CountriesFragment` or anything involving the details necessarily needs to change.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Going With One Activity

You might wonder why we need to bother with `DetailsActivity`. After all, the `EU4You` activity is perfectly capable of showing the `DetailsFragment` in a second pane — why not have it display the `DetailsFragment` in the first pane as well, in single-pane scenarios? Surely, this will be much simpler, as we can dispense with the activity and its entry in the manifest!

Yes, this is possible. No, it is not simpler.

The reason for the complexity is now managing all of our possible mix of fragments. We already had to deal with the following possibilities:

- Single-pane, showing the countries
- Single-pane, showing the countries, but on a large screen in portrait mode, after the activity had been launched in landscape, so the `DetailsFragment` exists in the `FragmentManager`, but is not visible
- Dual-pane, showing both fragments

If we get rid of `DetailsActivity` and dump all the responsibility onto `EU4You`, we have more scenarios:

- Single-pane, showing the details, having replaced the countries via a `FragmentTransaction`
- Single-pane, showing the countries, after having shown the details and the user then pressing BACK

Basically, what we must do now is `replace()` the `CountriesFragment` with the `DetailsFragment`, when we are in single-pane mode, when the user taps on a country in the list. This requires a fairly extensive number of changes, as you will see in the `LargeScreen/EU4YouSingleActivity` sample project.

The Revised Layouts

In our single-pane mode, our one pane will either hold the `CountriesFragment` or the `DetailsFragment`, depending upon what the user has done. Right now, our `FrameLayout` is named `R.id.countries`, which was fine before, but now seems like an inappropriate name. So, the new project's layouts change this to `R.id.mainfrag`, without changing anything else:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/mainfrag"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
/>
```

The New onCountrySelected()

The "simple" part of the changes comes in the revised `onCountrySelected()` method in EU4You:

```
  @Override
  public void onCountrySelected(Country c) {
    String url=getString(c.url);

    details.loadUrl(url);

    if (details.getId() != R.id.details) {
      getFragmentManager().beginTransaction()
                          .replace(R.id.mainfrag, details,
                                   TAG_DETAILS)
                          .addToBackStack(null).commit();
    }
  }
```

In our revised scenario, we will *always* have a `DetailsFragment`. The question is merely whether it is presently visible. Hence, we can call `loadUrl()` on `details` directly.

However, there are two possible scenarios for the status of our `DetailsFragment` at the point in time of `onCountrySelected()` being called:

1. It exists in the `details` `FrameLayout` of our dual-pane layout resource
2. It exists, perhaps due to a configuration change, but is not presently in a container

You might think that there would be a third scenario, where it is the visible fragment in the `mainfrag` `FrameLayout`. Indeed, sometimes `DetailsFragment` will be in that container... just not now. The only time that `onCountrySelected()` will be called is if the user tapped on an item in our `CountriesFragment`, which means that `CountriesFragment` must be in `mainfrag`.

The ID of a fragment, from `getId()`, is the ID of its container, when used with dynamic fragments. So, we check to see whether our `DetailsFragment` is in the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

details FrameLayout by comparing ID values. If they differ, then we commit() a replace() FragmentTransaction to put DetailsFragment into mainfrag. Note, though, that we use addToBackStack(), so if the user presses the BACK button, we will roll back this transaction and return to the CountriesFragment.

The New onCreate()

If you thought that was messy, you will not like the changes required to onCreate() of EU4You much more:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  countries=
      (CountriesFragment)getFragmentManager().findFragmentByTag(TAG_COUNTRIES);
  details=
      (DetailsFragment)getFragmentManager().findFragmentByTag(TAG_DETAILS);

  if (countries == null) {
    countries=new CountriesFragment();
    getFragmentManager().beginTransaction()
                                .add(R.id.mainfrag, countries,
                                     TAG_COUNTRIES).commit();
  }

  if (details == null) {
    details=new DetailsFragment();

    if (findViewById(R.id.details) != null) {
      getFragmentManager().beginTransaction()
                                  .add(R.id.details, details,
                                       TAG_DETAILS).commit();
    }
  }
  else {
    if (details.getId() == R.id.mainfrag) {
      if (findViewById(R.id.details) != null) {
        getFragmentManager().popBackStackImmediate();
      }
    }
    else {
      getFragmentManager().beginTransaction().remove(details)
                                  .commit();
    }

    if (findViewById(R.id.details) != null) {
      getFragmentManager().beginTransaction()
                                  .add(R.id.details, details,
                                       TAG_DETAILS).commit();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      }
    }
  }
```

This sample is derived from the original EU4You sample, and so we are still using a `FragmentTransaction` to set up the `CountriesFragment` in `mainfrag`, if we did not create `CountriesFragment` earlier.

Dealing with `DetailsFragment`, though, is decidedly more complicated. The flow that we want is if we were in dual-pane mode and switch to single-pane mode, that we show the `CountriesFragment` in that single pane. If we switch from single-pane mode to dual-pane mode, both fragments will be shown, of course.

First, we have the case where our `DetailsFragment` does not yet exist. This is much like the original sample: we need to create the fragment and put it into the `details` `FrameLayout`, if the `details` `FrameLayout` exists.

If the `DetailsFragment` exists, we need to make sure that it winds up in the `details` `FrameLayout`, if one exists.

To do that, we first check its ID to see if it is presently located in `mainfrag`. If it is, and if we have a `details` `FrameLayout`, we have switched to dual-pane mode and need to pop our back stack, in preparation for moving the `DetailsFragment` to the `details` `FrameLayout`.

If the `DetailsFragment` exists but is not in `mainfrag`, we `remove()` it entirely.

Then, if the `DetailsFragment` exists, regardless of where it was before, we `add()` it to the `details` `FrameLayout`.

The "OMG! Our Fragments Have No Views!" Changes

In testing, there are now scenarios in which `CountriesFragment` is called with `onSaveInstanceState()`, but without its views having been created (i.e., `onCreateView()` was not called). This would cause us to fail when trying to use `getListView()`, as that method would return `null`, since the `ListView` did not exist. So, we modify `onSaveInstanceState()` to be a bit more robust:

```
  @Override
  public void onSaveInstanceState(Bundle state) {
    super.onSaveInstanceState(state);

    if (getView() != null) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      state.putInt(STATE_CHECKED,
                   getListView().getCheckedItemPosition());
    }
  }
```

We also need to beef up `DetailsFragment` a bit. Before, we relied on the fact that, on a configuration change, our extras on our `Intent` for `DetailsActivity` would still be available. Now, though, there is no `DetailsActivity`, which means that `DetailsFragment` has to maintain its state, so that we do not lose the URL we were viewing when the user rotates the screen or causes another configuration change. And, to top it off, we have the same potential issue as with `CountriesFragment`, where the fragment might exist but not have `onCreateView()` called (e.g., we were in dual-pane mode and switched to single-pane mode, and `DetailsFragment` has not yet been displayed), so we cannot assume that `getWebView()` will always return a non-`null` value.

To that end, `DetailsFragment` gets complicated:

```
package com.commonsware.android.eu4you2;

import android.os.Bundle;
import android.webkit.WebViewFragment;

public class DetailsFragment extends WebViewFragment {
  private static final String STATE_URL="url";
  private String url=null;

  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);

    if (url == null && savedInstanceState != null) {
      url=savedInstanceState.getString(STATE_URL);
    }

    if (url != null) {
      loadUrl(url);
      url=null;
    }
  }

  @Override
  public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    if (url == null) {
      outState.putString(STATE_URL, getWebView().getUrl());
    }
    else {
      outState.putString(STATE_URL, url);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }

  void loadUrl(String url) {
    if (getView() == null) {
      this.url=url;
    }
    else {
      getWebView().loadUrl(url);
    }
  }
}
```

The `url` data member will temporarily hold the URL of the page we should be viewing, particularly when we have no `WebView` to work with. So, our `loadUrl()` method now puts the URL into `url` if we have no `WebView` or loads it into the `WebView` if the `WebView` exists. `onSaveInstanceState()` will put the URL — whether from `url` or from the `WebView` — into the state `Bundle`. `onActivityCreated()` will attempt to populate `url` from the `Bundle` (if we do not already have a URL), then use that to populate the `WebView` (which should exist if `onActivityCreated()` is called). `url` is set to `null` to indicate that the `WebView` holds our URL, once that is completed.

The Results

From a user experience standpoint, things have not significantly changed. The user still sees the list, still sees the details when tapping on an entry in the list, and still gets the dual-pane experience on larger screens.

However, the transition between the list and the details in single-pane mode is a bit faster, as a `FragmentTransaction` takes less time than does starting up another activity. However, by default, our `FragmentTransaction` does not apply any transition effects, and so the fragment changes "just happen" without any fades, zooms, or the like. It is certainly possible to specify fragment transition effects, if desired, though this is outside the scope of this chapter.

The Mashup Possibilities

It should be possible to combine the two revised versions of EU4You, having a single activity manage all the fragments, with `CountriesFragment` set up as a static fragment. The proof that this is possible is left to the reader.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The SlidingPaneLayout Variant

The R13 update to the Android Support package introduced `SlidingPaneLayout`, another way of handling this sort of master-detail pattern. `SlidingPaneLayout` significantly reduces the level of effort for setting up master-detail, as it handles all of the "dirty work" of showing the different fragments in different scenarios (normal screen, large screen, etc.).

Note that while the JavaDocs for `SlidingPaneLayout` contain "Experimental. This class may be removed", it is reasonably likely that somebody will continue maintaining this should Google abandon it.

The Role of SlidingPaneLayout

In the master-detail pattern, we are showing both the master and the detail fragment, side-by-side, on larger screens, while showing only one at a time on smaller screens. In the preceding examples, we had to manage all of that ourselves, in terms of deciding how many fragments to show and for switching between those fragments as needed.

`SlidingPaneLayout` encapsulates that logic.

`SlidingPaneLayout` will detect the screen size. If the screen size is big enough, `SlidingPaneLayout` will display its two children side-by-side. If the screen size is not big enough, `SlidingPaneLayout` will display one child at a time. However, by default, when the "master" child is visible, a thin strip on the right will allow the user to return to the "detail" child. Similarly, a swiping gesture can switch from the "detail" back to the "master" child. These are in addition to any changes in context you might introduce based on UI operations (e.g., tapping on an element in a master `ListView` automatically switching to the detail child).

Converting to SlidingPaneLayout

The `LargeScreen/EU4YouSlidingPane` sample project represents a rework of the EU4You core sample, this time using `SlidingPaneLayout` for handling the master-detail pattern.

Since `SlidingPaneLayout` encapsulates the master-detail logic, we can drop a lot of stuff that we used before but no longer need, including:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `DetailsActivity` (as `SlidingPaneLayout` works akin to our single-activity implementation)
- the dedicated large-screen layout (as `SlidingPaneLayout` "bakes in" the logic for handling different screen sizes)
- dynamic fragments (as `SlidingPaneLayout` will work better with static fragments, anyway)
- `isPersistentSelection()` (as we will always want to use activated rows, on API Level 11+, as the user can more readily switch back and forth between master and detail on smaller screens, and we want to indicate in the master what the context is that is displayed in the detail)

However, we did have to add a bit of pane management, plus move around some list-related behaviors in our `CountriesFragment`.

For starters, our `res/layout/main.xml` file now contains a `SlidingPaneLayout`, along with our two fragments, each set up as static `<fragment>` elements:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.SlidingPaneLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/panes"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <fragment
    android:id="@+id/countries"
    android:name="com.commonsware.android.eu4you4.CountriesFragment"
    android:layout_width="300sp"
    android:layout_height="match_parent"/>

  <fragment
    android:id="@+id/details"
    android:name="com.commonsware.android.eu4you4.DetailsFragment"
    android:layout_width="400dp"
    android:layout_height="match_parent"
    android:layout_weight="1"/>

</android.support.v4.widget.SlidingPaneLayout>
```

By putting an `android:layout_weight` on our `details` fragment, we indicate that we want that one to take up all remaining room when the two fragments are shown side-by-side. You might think that we should then set the width of the `details` fragment to `0dp`; however, for some reason, this does not work.

The size of the `countries` (master) fragment will be honored on larger screens. On smaller screens, the size of the master fragment will be dictated by the width of the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

screen, minus a strip to allow the user to see a portion of the detail fragment and swipe that to display the detail fragment *in toto*.

Our `CountriesFragment` now always sets up the `ListView` to be single-choice mode, in `onActivityCreated()`. It also calls `onCountrySelected()` on our `CountriesFragment.Contract`, to ensure that the master is highlighting the last selection — this is needed to make sure that everything is displayed properly after a configuration change:

```
  @Override
  public void onActivityCreated(Bundle state) {
    super.onActivityCreated(state);

    setListAdapter(new CountryAdapter());
    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);

    if (state != null) {
      int position=state.getInt(STATE_CHECKED, -1);

      if (position > -1) {
        getListView().setItemChecked(position, true);
        getContract().onCountrySelected(Country.EU.get(position));
      }
    }
  }
```

`onListItemClick()` of `CountriesFragment` becomes a bit simpler:

```
  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    l.setItemChecked(position, true);
    getContract().onCountrySelected(Country.EU.get(position));
  }
```

The `EU4You` activity overall becomes *substantially* simpler:

```
package com.commonsware.android.eu4you4;

import android.app.Activity;
import android.os.Bundle;
import android.support.v4.widget.SlidingPaneLayout;

public class EU4You extends Activity implements
    CountriesFragment.Contract {
  private DetailsFragment details=null;
  private SlidingPaneLayout panes=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    setContentView(R.layout.main);

    details=
        (DetailsFragment)getFragmentManager().findFragmentById(R.id.details);
    panes=(SlidingPaneLayout)findViewById(R.id.panes);
    panes.openPane();
  }

  @Override
  public void onBackPressed() {
    if (panes.isOpen()) {
      super.onBackPressed();
    }
    else {
      panes.openPane();
    }
  }

  @Override
  public void onCountrySelected(Country c) {
    details.loadUrl(getString(c.url));
    panes.closePane();
  }
}
```

In `SlidingPaneLayout` terminology, the pane is "open" if the *master* is shown on smaller screens, and the pane is "closed" if the *detail* is shown on smaller screens. If this feels a bit counter-intuitive to you, you are not alone in that regard.

By default, the `SlidingPaneLayout` is closed. So, if we want to start (on smaller screens) with the master pane shown, we need to call `openPane()`, as we do in `onCreate()`. Similarly:

- If we want to show the details when the user clicks on a country in the `CountriesFragment`, we need to call `closePane()` in `onCountrySelected()`
- If we want to show the master pane if the user presses BACK while viewing the detail pane, we need to override `onBackPressed()` and consume that event (calling `openPane()`), instead of performing the normal superclass behavior

What SlidingPaneLayout Looks Like

On a larger screen, the `SlidingPaneLayout` edition of the `EU4You` activity looks the same as the prior examples.

However, on a smaller screen, things look slightly different. Specifically:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Our master perspective has a thin strip on the right, showing a peek of the detail fragment

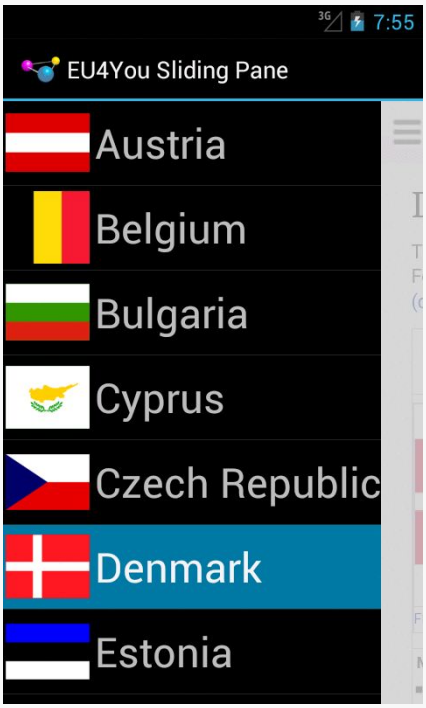


Figure 313: EU4YouSlidingPane, On a Phone Emulator, Showing Master

- The user can switch to the detail pane either by swiping open the detail pane or clicking on a country
- The user can switch back to the master pane either by swiping the detail pane back closed or by pressing the BACK button

Showing More Pages

`ViewPager` is a popular container in Android, as horizontal swiping is an increasingly popular navigational model, to move between peer pieces of content (e.g., swiping between contacts, swiping between book chapters). In some cases, when the `ViewPager` is on a larger screen, we simply want larger pages — a digital book reader, for example, would simply have a larger page in a bigger font for easier reading.

Sometimes, though, we might not be able to take advantage of the full space offered by the large screen, particularly when our `ViewPager` takes up the whole screen. In cases like this, it might be useful to allow `ViewPager`, in some cases, to show more

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

than one page at a time. Each "page" is then designed to be roughly phone-sized, and we choose whether to show one, two, or perhaps more pages at a time based upon the available screen space.

Mechanically, allowing `ViewPager` to show more than one page is fairly easy, involving overriding one more method in our `PagerAdapter`: `getPageWidth()`. To see this in action, take a look at the `ViewPager/MultiView1` sample project.

Each page in this sample is simply a `TextView` widget, using the activity's style's "large appearance", centered inside a `LinearLayout`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:gravity="center"
  android:orientation="vertical">

  <TextView
    android:id="@+id/text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

</LinearLayout>
```

The activity, in `onCreate()`, gets our `ViewPager` from the `res/layout/activity_main.xml` resource, and sets its adapter to be a `SampleAdapter`:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    pager=(ViewPager)findViewById(R.id.pager);
    pager.setAdapter(new SampleAdapter());
    pager.setOffscreenPageLimit(6);
  }
```

In this case, `SampleAdapter` is not a `FragmentPagerAdapter`, nor a `FragmentStatePagerAdapter`. Instead, it is its own implementation of the `PagerAdapter` interface:

```
  /*
   * Inspired by
   * https://gist.github.com/8cbe094bb7a783e37ad1
   */
  private class SampleAdapter extends PagerAdapter {
    @Override
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    public Object instantiateItem(ViewGroup container, int position) {
      View page=
          getLayoutInflater().inflate(R.layout.page, container, false);
      TextView tv=(TextView)page.findViewById(R.id.text);
      int blue=position * 25;

      final String msg=
          String.format(getString(R.string.item), position + 1);

      tv.setText(msg);
      tv.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
          Toast.makeText(MainActivity.this, msg, Toast.LENGTH_LONG)
               .show();
        }
      });

      page.setBackgroundColor(Color.argb(255, 0, 0, blue));
      container.addView(page);

      return(page);
    }

    @Override
    public void destroyItem(ViewGroup container, int position,
                            Object object) {
      container.removeView((View)object);
    }

    @Override
    public int getCount() {
      return(9);
    }

    @Override
    public float getPageWidth(int position) {
      return(0.5f);
    }

    @Override
    public boolean isViewFromObject(View view, Object object) {
      return(view == object);
    }
  }
```

To create your own `PagerAdapter`, the big methods that you need to implement are:

- `instantiateItem()`, where you create the page itself and add it to the supplied container. In this case, we inflate the page, set the text of the `TextView` based on the supplied position, set the background color of the page itself to be a different shade of blue based on the position, set up a click

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

listener to show a `Toast` when the `TextView` is tapped, and use that for our page. We return some object that identifies this page; in this case, we return the inflated `View` itself. A fragment-based `PagerAdapter` would probably return the fragment.

- `destroyItem()`, where we need to clean up a page that is being removed from the pager, where the page is identified by the `Object` that we had previously returned from `instantiateItem()`. In our case, we just remove it from the supplied container.
- `isViewFromObject()`, where we confirm whether some specific page in the pager (represented by a `View`) is indeed tied to a specific `Object` returned from `instantiateItem()`. In our case, since we return the `View` from `instantiateItem()`, we merely need to confirm that the two objects are indeed one and the same.
- `getCount()`, as with the built-in `PagerAdapter` implementations, to return how many total pages there are.

In our case, we also override `getPageWidth()`. This indicates, for a given position, how much horizontal space in the `ViewPager` should be given to this particular page. In principle, each page could have its own unique size. The return value is a `float`, from `0.0f` to `1.0f`, indicating what fraction of the pager's width goes to this page. In our case, we return `0.5f`, to have each page take up half the pager.

The result is that we have two pages visible at a time:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

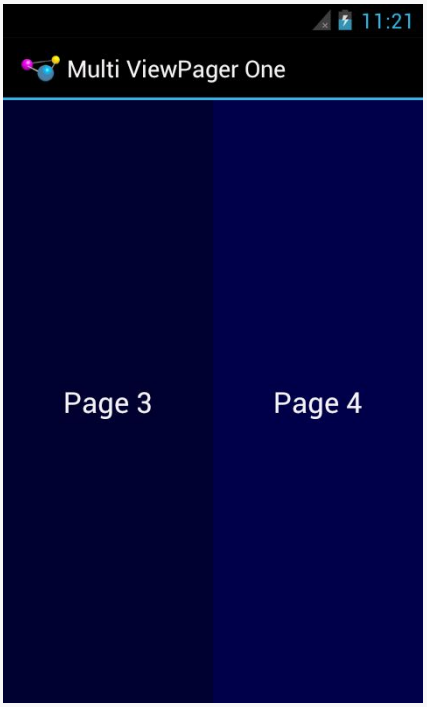


Figure 314: Two Pages in a ViewPager on Android 4.0.3

It is probably also a good idea to call `setOffscreenPageLimit()` on the `ViewPager`, as we did in `onCreate()`. By default (and at minimum), `ViewPager` will cache three pages: the one presently visible, and one on either side. However, if you are showing more than one at a time, you should bump the limit to be 3 times the number of simultaneous pages. For a page width of `0.5f` — meaning two pages at a time – you would want to call `setOffscreenPageLimit(6)`, to make sure that you had enough pages cached for both the current visible contents and one full swipe to either side.

`ViewPager` even handles "partial swipes" — a careful swipe can slide the right-hand page into the left-hand position and slide in a new right-hand page. And `ViewPager` stops when you run out of pages, so the last page will always be on the right, no matter how many pages at a time and how many total pages you happen to have.

The biggest downside to this approach is that it will not work well with the current crop of indicators. `PagerTitleStrip` and `PagerTabStrip` assume that there is a single selected page. While the indicator will adjust properly, the visual representation shows that the left-hand page is the one selected (e.g., the tab with the highlight), even though two or more pages are visible. You can probably overcome this with a custom indicator (e.g., highlight the selected tab and the one to its right).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Also note that this approach collides a bit with `setPageMargin()` on `ViewPager`. `setPageMargin()` indicates an amount of whitespace that should go in a gutter between pages. In principle, this would work great with showing multiple simultaneous pages in a `ViewPager`. However, `ViewPager` does not take the gutter into account when interpreting the `getPageWidth()` value. For example, suppose `getPageWidth()` returns `0.5f` and we `setPageMargin(20)`. On a 480-pixel-wide `ViewPager`, we will actually use 500 pixels: 240 for the left page, 240 for the right page, and 20 for the gutter. As a result, 20 pixels of our right-hand page are off the edge of the pager. Ideally, `ViewPager` would subtract out the page margin *before* applying the page width. One workaround is for you to derive the right `getPageWidth()` value based upon the `ViewPager` size and gutter yourself, rather than hard-coding a value. Or, build in your gutter into your page contents (e.g., using `android:layout_marginLeft` and `android:layout_marginRight`) and skip `setPageMargin()` entirely.

Columns or Pages

Another pattern — using pages for smaller screens and having the "pages" side-by-side in columns for larger screens — will be explored later in the book.

The Grid Pattern

Yet another approach for taking advantage of larger screen sizes is to always show a full-size master and a full-size detail — perhaps using different activities — but to use a grid rather than a list for the master. This works well when the data being shown in the grid can be represented as "cards", often dominated by some photo or other image.

The basic approach is to use fewer grid columns (e.g., 1 or 2) on smaller screen sizes and more grid columns (e.g., 3 or 4) on larger screen sizes. This way, the application flow is identical across screen sizes, yet the screen usage on larger screens is more effective. This is particularly true if you use on of the "staggered" grid widgets available from third parties, like Etsy's `AndroidStaggeredGrid` or Maurycy Wojtowicz's `StaggeredGridView`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 315: StaggeredGridView Demo (image courtesy of Maurycy Wojtowicz)

Fragment FAQs

Here are some other common questions about the use of fragments in support of large screen sizes:

Does *Everything* Have To Be In a Fragment?

In a word, no.

UI constructs that do not change based on screen size, configurations, and the like could simply be defined in the activity itself. For example, the activity can add items to the action bar that should be there regardless of what fragments are shown.

What If Fragments Are Not Right For Me?

While fragments are useful, they do not solve all problems. Few games will use fragments for the core of game play, for example. Applications with other forms of specialized user interfaces — painting apps, photo editors, etc. – may also be better served by eschewing fragments *for those specific activities* and doing something else.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That "something else" might start with custom layouts for the different sizes and orientations. At runtime, you can determine what you need either by inspecting what you got from the layout, or by using `Configuration` and `DisplayMetrics` objects to determine what the device capabilities are (e.g., screen size). The activity would then need to have its own code for handling whatever you want to do differently based on screen size (e.g., offering a larger painting canvas plus more on-screen tool palettes).

Do Fragments Work on TVs?

Much of the focus on "larger-screen devices" has been on tablets, because, as of the time of this writing, they are the most popular "larger-screen devices" in use. However, there are plenty of scenarios involving Android on TV to consider. A TV or other external display will presents itself as a `-large` (720p) or `-xlarge` (1080p) screen. Fragments can certainly help with displaying a UI for a TV, but there are other design considerations to take into account, based upon the fact that the user sits much further from a TV than they do from a phone or tablet (so-called "10-foot user experience"). This is covered in greater detail later in the book.

Screen Size and Density Tactics

Even if we take the "tablet = several phones" design approach, the size of the "phone" will vary, depending on the size of the tablet. Plus, there are real actual *phones*, and those too vary in size. Hence, our fragments (or activities hosting their own UI directly) need to take into account micro fluctuations in size, as well as the macro ones.

Screen density is also something that affects us tactically. It is rare that an application will make wholesale UI changes based upon whether the screen is 160dpi or 240dpi or 320dpi or something else. However, changes in density can certainly impact the sizes of things, like images, that are intrinsically tied to pixel sizes. So, we need to take density into account as we are crafting our fragments to work well in a small range of sizes.

Dimensions and Units

As a unit of measure, the pixel (`px`) is a poor choice, because its size varies by density. Two phones might have very similar screen sizes but radically different densities. Anything specified in terms of pixels will be smaller on the higher-density device, and typically you would want them to be about the same size. For example, a

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`Button` should not magically shrink for a ~4" phone just because the phone happens to have a much higher screen density than some other phone.

The best answer is to avoid specifying concrete sizes where possible. This is why you tend to see containers, and some widgets, use `match_parent` and `wrap_content` for their size — those automatically adjust based upon device characteristics.

Some places, though, you have to specify a more concrete size, such as with padding or margins. For these, you have two major groups of units of measure to work with:

- Those based upon pixels, but taking device characteristics into account. These include density-independent pixels (`dp` or `dip`), which try to size each `dp` to be about 1/160 of an inch. These also include scaled pixels (`sp`), which scales the size based upon the default font size on the device — `sp` is often used with `TextView` (and subclasses) for `android:textSize` attributes.
- Those based purely on physical units of measure: `mm` (millimeters), `in` (inches), and `pt` (points = 1/72 of an inch).

Any of those tends to be better than `px`. Which you choose will depend on which you and your graphics designer are more comfortable with.

If you find that there are cases where the dimensions you want to use vary more widely than the automatic calculations from these density-aware units of measure, you can use dimension resources. Create a `dimens.xml` file in `res/values/` and related resource sets, and put in there `<dimen>` elements that give a dimension a name and a size. In addition to perhaps making things a bit more DRY ("don't repeat yourself"), you can perhaps create different values of those dimensions for different screen sizes, densities, or other cases as needed.

Layouts and Stretching

Web designers need to deal with the fact that the user might resize their browser window. The approaches to deal with this are called "fluid" designs.

Similarly, Android developers need to create "fluid" layouts for fragments, rows in a `ListView`, and so on, to deal with similar minor fluctuations in size.

Each of "The Big Three" container classes has its approach for dealing with this:

- Use `android:layout_weight` with `LinearLayout` to allocate extra space

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Use `android:stretchColumns` and `android:shrinkColumns` with `TableLayout` to determine which columns should absorb extra space and which columns should be forcibly "shrunk" to yield space for other columns if we lack sufficient horizontal room
- Use appropriate rules on `RelativeLayout` to anchor widgets as needed to other widgets or the boundaries of the container, such that extra room flows naturally wherever the rules call for

Drawables That Resize

Images, particularly those used as backgrounds, will need to be resized to take everything into account:

- screen size and density
- size of the widget, and its contents, for which it serves as the background (e.g., amount of prose in a `TextView`)

Android supports what is known as the "nine-patch" PNG format, where resizing information is held in the PNG itself. This is typically used for things like rounded rectangles, to tell Android to stretch the straight portions of the rectangle but to not stretch the corners. Nine-patch PNG files will be examined in greater detail in a later chapter of this book.

The `ShapeDrawable` XML drawable resource uses an ever-so-tiny subset of SVG (Scalable Vector Graphics) to create a vector art definition of an image. Once again, this tends to be used for rectangles and rounded rectangles, particularly those with a gradient fill. Since Android interprets the vector art definition at runtime, it can create a smooth gradient, interpolating all intervening colors from start to finish. Stretching a PNG file — even a nine-patch PNG file — tends to result in "banding effects" on the gradients. `ShapeDrawable` is also covered later in this book.

Third-party libraries can also help. The `svg-android` project supplies a JAR that handles more SVG capabilities than does `ShapeDrawable`, though it too does not cover the entire SVG specification. Also, `WebView` has some ability to view SVG files on Android 3.0+.

Drawables By Density

Sometimes, though, there is no substitute for your traditional bitmap image. Icons and related artwork are not necessarily going to be stretched at runtime, but they are still dependent upon screen density. A 80x80 pixel image may look great on a

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Samsung Galaxy Nexus or other `-xhdpi` device, coming in at around a quarter-inch on a side. However, when viewed on a `-mdpi` device, that same icon will be a half-inch on a side, which may be entirely too large.

The best answer is to create multiple renditions of the icon at different densities, putting each icon in the appropriate drawable resource directory (e.g., `res/drawable-mdpi`, `res/drawable-hdpi`). This is what Android Asset Studio did for us in the tutorials, creating launcher icons from some supplied artwork for all four densities. Even better is to create icons tailored for each density — rather than just reducing the pixel count, take steps to draw an icon that will still make sense to the user at the lower pixel count, exaggerating key design features and dropping other stuff off. Google's Kiril Grouchnikov has an excellent blog post on this aspect

However, Android will let you cheat.

If you supply only some densities, but your app runs on a device with a different density, Android will automatically resample your icons to try to generate one with the right density, to keep things the same size. On the plus side, this saves you work — perhaps you only ship an `-xhdpi` icon and let Android do the rest. And it can reduce your APK size by a bit. However, there are costs:

- This is a bit slower at runtime and consumes a bit more battery
- Android's resampling algorithm may not be as sophisticated as that of your preferred image editor (e.g., Photoshop)
- You cannot finesse the icon to look better than a simple resampling (e.g., drop off design elements that become unidentifiable)

Other Considerations

There are other things you should consider when designing your app to work on multiple screen sizes, beyond what is covered above.

Small-Screen Devices

It is easy to think of screen size issues as being "phones versus tablets". However, not only do tablets come in varying sizes (5" Samsung Galaxy Note to a bunch of 10.1" tablets), but phones come in varying sizes. Those that have less than a 3" diagonal screen size will be categorized as `-small` screen devices, and you can have different layouts for those.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Getting things to work on small screens is sometimes more difficult than moving from normal to larger screens, simply because you lack sufficient room. You can only shrink widgets so far before they become unreadable or "untappable". You may need to more aggressively use `ScrollView` to allow your widgets to have more room, but requiring the user to pan through your whole fragment's worth of UI. Or, you may need to divide your app into more fragments than you originally anticipated, and use more activities or other tricks to allow the user to navigate the fragments individually on small-screen devices, while stitching them together into larger blocks for larger phones.

Avoid Full-Screen Backgrounds

Android runs in lots of different resolutions.

Lots and lots of different resolutions.

Trying to create artwork for each and every resolution in use today will be tedious and fragile, the latter because new resolutions pop up every so often, ones you may not be aware of.

Hence, try to design your app to avoid some sort of full-screen background, where you are expecting the artwork to perfectly fit the screen. Either:

- Do not use a background, or
- Use a background, but one that is designed to be cropped to fit and will look good in its cropped state, or
- Use a background, but one that can naturally bleed into some solid fill to the edges (e.g., a starfield that simply lacks stars towards the edges), so you can "fill in" space around your background with that solid color to fill the screen, or
- Dynamically draw the background (e.g., a starfield where you place the stars yourself at runtime using 2D graphics APIs)

For most conventional apps, just using the background from your stock theme will typically suffice. This problem is much bigger for 2D games, which tend to rely upon backgrounds as a game surface.

Manifest Elements for Screen Sizes

There are two elements you can add to your manifest that impact how your application will behave with respect to screen sizes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

<compatible-screens> serves as an advertisement of your capabilities, to the Google Play Store and similar "markets". You can have a <compatible-screens> element with one or more child <screen> elements — each <screen> enumerates a combination of screen size and screen density that you support:

```
<compatible-screens>
    <!-- all possible normal size screens -->
    <screen android:screenSize="normal" android:screenDensity="ldpi" />
    <screen android:screenSize="normal" android:screenDensity="mdpi" />
    <screen android:screenSize="normal" android:screenDensity="hdpi" />
    <screen android:screenSize="normal" android:screenDensity="xhdpi" />
    <!-- all possible large size screens -->
    <screen android:screenSize="large" android:screenDensity="ldpi" />
    <screen android:screenSize="large" android:screenDensity="mdpi" />
    <screen android:screenSize="large" android:screenDensity="hdpi" />
    <screen android:screenSize="large" android:screenDensity="xhdpi" />
</compatible-screens>
```

The Google Play Store will filter your app, so it will not show up on devices that have screens that do not meet one of your <screen> elements. However, new densities show up every year or so, and devices running those densities will *not* be supported by your <compatible-screens> element unless you add the appropriate <screen> element for that density. For example, the above <compatible-screens> element does not cover tvdpi or xxhdpi devices. As a result, Google discourages the use of <compatible-screens>.

There is also a <supports-screens> element, as we saw when we set up our initial project in the tutorials. Here, you indicate what screen sizes you support, akin to <compatible-screens> (minus any density declarations). And, the Google Play Store will filter your app, so it will not show up on devices that have screens *smaller than what you support*.

So, for example, suppose that you have a <supports-screens> element like this:

```
<supports-screens android:smallScreens="false"
                  android:normalScreens="true"
                  android:largeScreens="true"
                  android:xlargeScreens="false"
/>
```

You will not show up in the Google Play Store for any -small screen devices. However, you *will* show up in the Google Play Store for any -xlarge screen devices — Android will merely apply some runtime logic to try to help your app run well on such screens. So, while <compatible-screens> is purely a filter, <supports-screens>

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

is a filter for smaller-than-supported screens, and a runtime "give me a hand!" flag for larger-than-supported screens.

Considering Newer Densities

`-tvdpi` — around 213dpi — was added for Android TV, and is the density used for 720p Android TV devices. However, Google *also* elected to use `-tvdpi` for the Nexus 7 tablet. However, not even Google bothered to create many `-tvdpi`-specific resources, allowing the OS to downsample from the `-hdpi` edition.

`-xxhdpi` was added in late 2012 and is for devices with a screen density around 480dpi. While Android can up-sample an `-xhdpi` image for `-xxhdpi`, the results may not be as crisp as you would like. Hence, you may wish to consider creating `-xxhdpi` as your "top tier" density, so other devices can downsample if needed. At the time of this writing, about 15% of Play Store-equipped Android devices are `-xxhdpi`.

`-xxxhdpi` is for devices with screens around 640dpi. At the time of this writing `-xxxhdpi` is not in significant use.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tutorial #19 - Supporting Large Screens

So far, we have created a variety of fragments that are being used one at a time in a hosting activity: notes, help, and about. And, on smaller-screen devices, like phones, that is probably the best solution. But on devices like 10" tablets, it might be nice to be able to have some of those fragments take over a part of the main activity's space. For example, the user could be reading the chapter and reading the online help.

Hence, in this tutorial, we will arrange for the help and about fragments to be loaded into `EmPubLiteActivity` directly on tablets, while retaining our existing functionality for other devices.

This is a continuation of the work we did in the previous tutorial.

You can find the results of the previous tutorial and the results of this tutorial in the book's GitHub repository:

- Android Studio previous tutorial and this tutorial
- Eclipse previous tutorial and this tutorial

Step #1: Creating Our Layouts

The simplest way to both add a place for these other fragments and to determine when we should be using these other fragments in the main activity is to create new `layout` resource sets for larger-screen devices, with customized versions of `main.xml` to be used by `EmPubLiteActivity`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The following sections contain instructions for Android Studio and Eclipse — choose the instructions appropriate for the IDE you are using for these tutorials.

Android Studio

Right-click over the `res/` directory in your app, then choose New > "Android Resource Directory" from the context menu. As before, this brings up the new resource directory dialog:

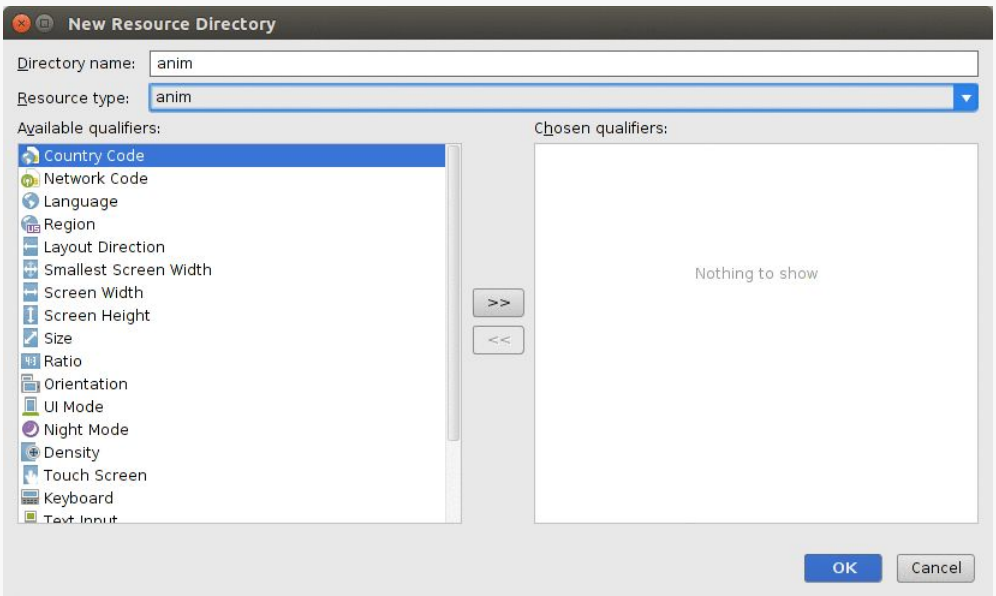


Figure 316: Android Studio New Resource Dialog, As Initially Opened

Choose "layout" from the "Resource type" drop-down. Then, click on "Screen Width" in the list of qualifiers on the left, and click the ">>" button to add that to the list on the right:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

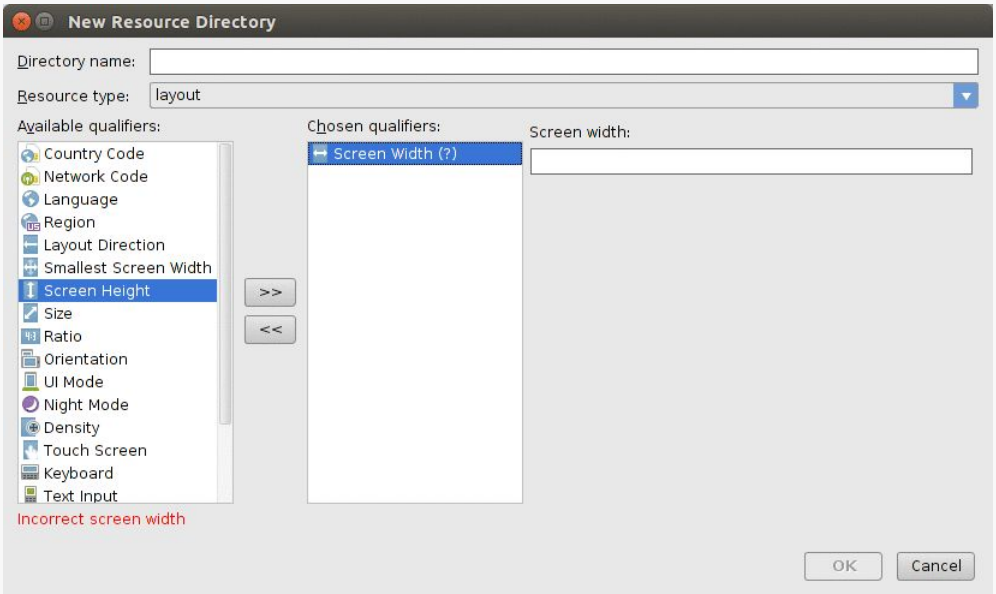


Figure 317: Android Studio New Resource Dialog, After Selecting "Screen Width"

In the "Screen width" field, fill in `880`:

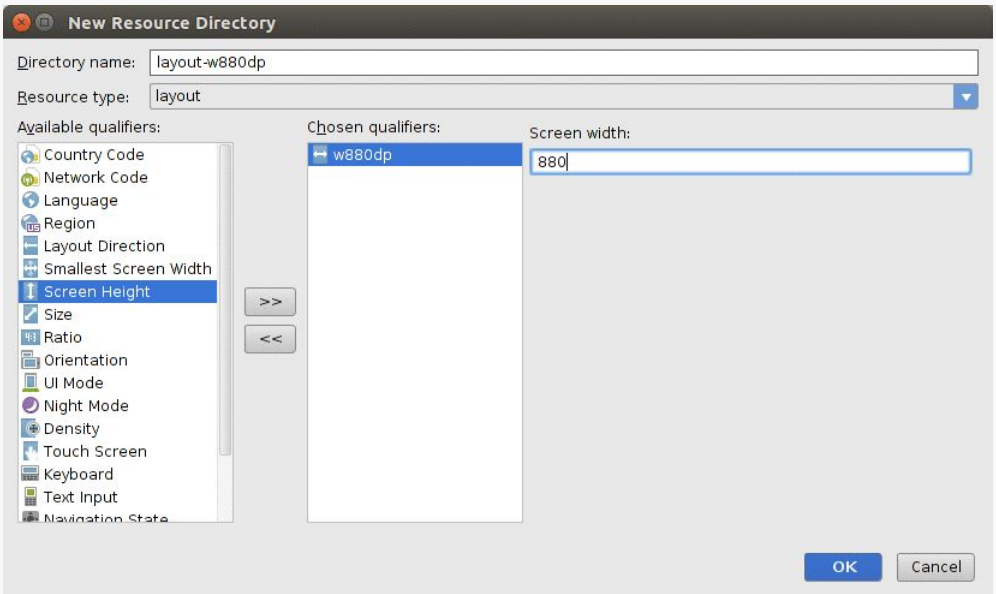


Figure 318: Android Studio New Resource Dialog, After Setting Screen Width

Click OK to create the directory. Repeat that process to create a `res/layout-h880dp/` directory, this time choosing "Screen Height" rather than "Screen Width".

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Then, right-click over the `res/layout/main.xml` file and choose "Copy" from the context menu. After that, right-click over the new `res/layout-w880dp/` directory and choose "Paste" from the context menu. This brings up the copy dialog:

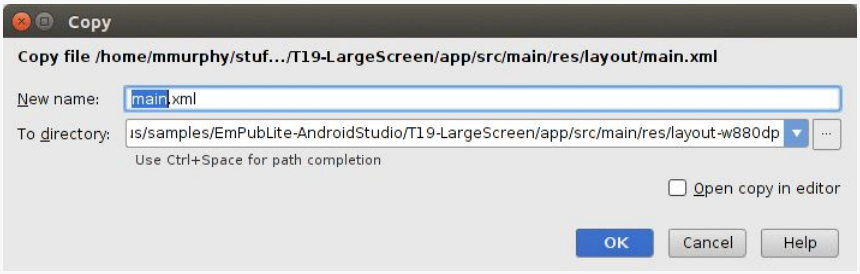


Figure 319: Android Studio Copy Dialog

Check the "Open copy in editor" checkbox and click OK. This will bring up the graphical layout editor on this copy of the `main` layout.

Unfortunately, what we want to do is not readily supported by Android Studio's edition of the drag-and-drop GUI builder. So, switch over to the XML for this layout, and replace it with:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/something"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <RelativeLayout
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="7"
    tools:context=".EmPubLiteActivity">

    <ProgressBar
      android:id="@+id/progressBar1"
      style="?android:attr/progressBarStyleLarge"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_centerHorizontal="true"
      android:layout_centerVertical="true"/>

    <android.support.v4.view.ViewPager
      android:id="@+id/pager"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:visibility="gone"/>
  </RelativeLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <View
    android:id="@+id/divider"
    android:layout_width="2dp"
    android:layout_height="match_parent"
    android:background="#AA000000"
    android:visibility="gone"/>

  <FrameLayout
    android:id="@+id/sidebar"
    android:layout_width="0dp"
    android:layout_height="match_parent">

  </FrameLayout>

</LinearLayout>
```

Repeat the same process, copying `res/layout/main.xml` into the `res/layout-h880dp/` directory, and replacing the copy's contents with:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/something"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="7"
    tools:context=".EmPubLiteActivity">

    <ProgressBar
      android:id="@+id/progressBar1"
      style="?android:attr/progressBarStyleLarge"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_centerHorizontal="true"
      android:layout_centerVertical="true"/>

    <android.support.v4.view.ViewPager
      android:id="@+id/pager"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:visibility="gone"/>
  </RelativeLayout>

  <View
    android:id="@+id/divider"
    android:layout_width="match_parent"
    android:layout_height="2dp"
    android:background="#AA000000"
    android:visibility="gone"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <FrameLayout
    android:id="@+id/sidebar"
    android:layout_width="match_parent"
    android:layout_height="0dp">
  </FrameLayout>

</LinearLayout>
```

Eclipse

First, right-click over the `res/` folder, and choose New > Folder from the context menu. Fill in `layout-w880dp` as the folder name, then click "Finish" to create the folder. This resource set will be used on devices where the current width (given the current orientation) is 880dp or wider (a.k.a., 5.5" wide or wider).

Then, right-click over the `res/layout/main.xml` file and choose "Copy" from the context menu. After that, right-click over the new `res/layout-w880dp/` folder and choose "Paste" from the context menu. This makes a copy of your `main.xml` resource that we can use for `-w880dp` devices.

Double-click on the `res/layout-w880dp/main.xml` file to bring it up in the graphical layout editor. In the Outline pane, right-click on the `RelativeLayout` and choose "Wrap in Container..." from the context menu. Choose "LinearLayout (horizontal)" in the drop-down list of available containers, and give the container some ID (the value does not matter, as we will not be using it, but the dialog requires it). Click OK to wrap our `RelativeLayout` in the horizontal `LinearLayout`.

Click on the `RelativeLayout` in the Outline pane. In the Properties pane, in the "Layout Parameters" group, fill in `7` in the "Weight" field, and fill in `0dp` for the "Width".

In the Palette, switch to the Advanced group of widgets, and drag a `View` over to the Outline pane and drop it on the `LinearLayout`, which will add it to the end of the `LinearLayout` roster of children. You may find it easier to drop it on the `LinearLayout` in the Outline view. Make the following adjustments to the properties of the `View` using the Properties pane:

- Set the Id to `@+id/divider`
- Set the Height to `match_parent`
- Set the Width to `2dp`
- Set the Background to `#AA000000`
- Set the Visibility to `gone`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Then, switch over to the XML, and if you see an erroneous `android:layout_weight` attribute on this `View`, get rid of it.

Back in the Palette, switch to the Layouts group of widgets, and drag a `FrameLayout` over to the Outline pane and drop it on the `LinearLayout`, adding it as a third child. Make the following adjustments to the properties of the `FrameLayout` using the Properties pane:

- Set the Id to `@+id/sidebar`
- Set the Weight to `0`
- Set the Width to `0dp`

Save your changes (e.g., `<Ctrl>-<S>`).

Then, right-click over the `res/layout-w880dp` folder, and choose Copy from the context menu.

Next, right-click over the `res/` folder, and choose Paste from the context menu. When the dialog appears asking for a name for the pasted copy, set it to `layout-h880dp`. This folder's resources will be used on devices where the current *height* is 880dp.

Double-click on `res/layout-h880dp/main.xml` file, to bring it up in the graphical layout editor. Click on the `LinearLayout` and, in the Properties pane, set the "Orientation" to be `vertical`.

Then, switch over to the XML view, and swap the `android:layout_width` and `android:layout_height` values for the `RelativeLayout`, the `View`, and the `FrameLayout`. When you are done, each should have an `android:layout_width` of `match_parent` and an `android:layout_height` of `0dp` (except the `View`, which should be `2dp`).

Save your changes (e.g., `<Ctrl>-<S>`).

Step #2: Loading Our Sidebar Widgets

Now that we added the divider widget and sidebar container to (some of) our layouts, we need to access those widgets at runtime.

So, in `EmPubLiteActivity`, add data members for them:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private View sidebar=null;
  private View divider=null;
```

Then, in `onCreate()` of `EmPubLiteActivity`, initialize those data members, sometime after the call to `setContentView()`:

```
    sidebar=findViewById(R.id.sidebar);
    divider=findViewById(R.id.divider);
```

Step #3: Opening the Sidebar

A real production-grade app would use animated effects to hide and show our sidebar. However, we have not yet covered animations in this book, so we will simply:

- Cause the divider to become visible
- Adjust the `android:layout_weight` of our sidebar to be `3` instead of `0`, giving it ~30% of the screen (with the original `RelativeLayout` getting 70%, courtesy of its `android:layout_weight="7"`)

With that in mind, add the following implementation of an `openSidebar()` method to `EmPubLiteActivity`:

```
  private void openSidebar() {
    LinearLayout.LayoutParams p=
        (LinearLayout.LayoutParams)sidebar.getLayoutParams();
    if (p.weight == 0) {
      p.weight=3;
      sidebar.setLayoutParams(p);
    }
    divider.setVisibility(View.VISIBLE);
  }
```

Here, we:

- Get the existing `LinearLayout.LayoutParams` from the sidebar
- If it is still `0` (meaning the sidebar has not been opened), assign it a weight of `3`, update the layout via `setLayoutParams()`, and toggle the visibility of the divider

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Step #4: Loading Content Into the Sidebar

Now that we can get our sidebar to appear, we need to load content into it... but only if we have the sidebar. If `EmPubLiteActivity` loads a layout that does not have the sidebar, we need to stick with our existing logic that starts up an activity to display the content.

With that in mind, add data members to `EmPubLiteActivity` to hold onto our help and about fragments:

```
  private SimpleContentFragment help=null;
  private SimpleContentFragment about=null;
```

Also add a pair of static data members that will be used as tags for identifying these fragments in our `FragmentManager`:

```
  private static final String HELP="help";
  private static final String ABOUT="about";
```

Also add a pair of static data members that will hold the paths to our help and about assets, since we will be referring to them from more than one place when we are done:

```
  private static final String FILE_HELP=
      "file:///android_asset/misc/help.html";
  private static final String FILE_ABOUT=
      "file:///android_asset/misc/about.html";
```

In `onCreate()` of `EmPubLiteActivity`, initialize the fragments from the `FragmentManager`:

```
    help=
        (SimpleContentFragment)getFragmentManager().findFragmentByTag(HELP);
    about=
        (SimpleContentFragment)getFragmentManager().findFragmentByTag(ABOUT);
```

The net result is that *if* we are returning from a configuration change, we will have our fragments, otherwise we will not at this point.

Next, add the following methods to `EmPubLiteActivity`:

```
  private void showAbout() {
    if (sidebar != null) {
      openSidebar();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      if (about == null) {
        about=SimpleContentFragment.newInstance(FILE_ABOUT);
      }

      getFragmentManager().beginTransaction().addToBackStack(null)
          .replace(R.id.sidebar, about).commit();
    }
    else {
      Intent i=new Intent(this, SimpleContentActivity.class);

      i.putExtra(SimpleContentActivity.EXTRA_FILE, FILE_ABOUT);
      startActivity(i);
    }
  }

  private void showHelp() {
    if (sidebar != null) {
      openSidebar();

      if (help == null) {
        help=SimpleContentFragment.newInstance(FILE_HELP);
      }

      getFragmentManager().beginTransaction().addToBackStack(null)
          .replace(R.id.sidebar, help).commit();
    }
    else {
      Intent i=new Intent(this, SimpleContentActivity.class);

      i.putExtra(SimpleContentActivity.EXTRA_FILE, FILE_HELP);
      startActivity(i);
    }
  }
```

Both of these methods follows the same basic recipe:

- Check to see if `sidebar` is `null`, to see if we have a sidebar or not
- If we have a sidebar, call `openSidebar()` to ensure the user can see the sidebar, create our `Fragment` if we do not already have it, and use a `FragmentTransaction` to replace whatever was in the sidebar with the new `Fragment`
- If we do not have the sidebar, launch an activity with an appropriately-configured `Intent`

Note a couple of things with our `FragmentTransaction` objects:

- We use `addToBackStack(null)`, so if the user presses BACK, Android will reverse this transaction

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- We use `replace()` instead of `add()`, as there may already be a fragment in the sidebar (`replace()` will behave the same as `add()` for an empty sidebar)

Then, in the `onOptionsItemSelected()` of `EmPubLiteActivity`, replace the `about`, and `help` `case` blocks to use the newly-added methods, replacing their existing implementations:

```
      case R.id.about:
        showAbout();

        return(true);

      case R.id.help:
        showHelp();

        return(true);
```

Step #5: Removing Content From the Sidebar

While `addToBackStack(null)` will allow Android to automatically remove fragments as the user presses BACK, that will not cause our sidebar to magically close. Rather, we need to do that ourselves.

The easiest way to track this is to track the state of the "back stack". So, add `implements FragmentManager.OnBackStackChangedListener` to the declaration of `EmPubLiteActivity`, and in `onCreate()` of `EmPubLiteActivity`, add the following line, sometime after you initialized the `sidebar` and `divider` data members:

```
    getFragmentManager().addOnBackStackChangedListener(this);
```

The first statement registers our activity as receiving events related to changes in the state of the back stack.

To make this compile, we need to implement `onBackStackChanged()` in `EmPubLiteActivity`:

```
  @Override
  public void onBackStackChanged() {
    if (getFragmentManager().getBackStackEntryCount() == 0) {
      LinearLayout.LayoutParams p=
          (LinearLayout.LayoutParams)sidebar.getLayoutParams();
      if (p.weight > 0) {
        p.weight=0;
        sidebar.setLayoutParams(p);
        divider.setVisibility(View.GONE);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      }
    }
  }
```

Here, if our back stack is empty, we reverse the steps from `openSidebar()` and close it back up again, hiding the divider and setting the sidebar's weight to `0`.

The complete revised `EmPubLiteActivity` should now look something like:

```
package com.commonsware.empublite;

import android.app.Activity;
import android.app.FragmentManager;
import android.content.Intent;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.os.StrictMode;
import android.support.v4.view.ViewPager;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.LinearLayout;
import com.commonsware.cwac.wakeful.WakefulIntentService;
import de.greenrobot.event.EventBus;

public class EmPubLiteActivity extends Activity implements
    FragmentManager.OnBackStackChangedListener {
  private static final String MODEL="model";
  private static final String PREF_LAST_POSITION="lastPosition";
  private static final String PREF_SAVE_LAST_POSITION="saveLastPosition";
  private static final String PREF_KEEP_SCREEN_ON="keepScreenOn";
  private static final String HELP="help";
  private static final String ABOUT="about";
  private static final String FILE_HELP=
      "file:///android_asset/misc/help.html";
  private static final String FILE_ABOUT=
      "file:///android_asset/misc/about.html";
  private ViewPager pager=null;
  private ContentsAdapter adapter=null;
  private ModelFragment mfrag=null;
  private View sidebar=null;
  private View divider=null;
  private SimpleContentFragment help=null;
  private SimpleContentFragment about=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setupStrictMode();

    setContentView(R.layout.main);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    pager=(ViewPager)findViewById(R.id.pager);
    sidebar=findViewById(R.id.sidebar);
    divider=findViewById(R.id.divider);

    getFragmentManager().addOnBackStackChangedListener(this);

    help=
        (SimpleContentFragment)getFragmentManager().findFragmentByTag(HELP);
    about=
        (SimpleContentFragment)getFragmentManager().findFragmentByTag(ABOUT);

    getActionBar().setHomeButtonEnabled(true);
    UpdateReceiver.scheduleAlarm(this);
  }

  @Override
  public void onResume() {
    super.onResume();

    EventBus.getDefault().register(this);

    if (adapter==null) {
      mfrag=
          (ModelFragment)getFragmentManager().findFragmentByTag(MODEL);

      if (mfrag == null) {
        mfrag=new ModelFragment();
        getFragmentManager().beginTransaction().add(mfrag, MODEL).commit();
      }
      else if (mfrag.getBook() != null) {
        setupPager(mfrag.getBook());
      }
    }

    if (mfrag.getPrefs() != null) {
      pager.setKeepScreenOn(mfrag.getPrefs()
          .getBoolean(PREF_KEEP_SCREEN_ON, false));
    }
  }

  @Override
  public void onPause() {
    EventBus.getDefault().unregister(this);

    if (mfrag.getPrefs() != null) {
      int position=pager.getCurrentItem();

      mfrag.getPrefs().edit().putInt(PREF_LAST_POSITION, position)
          .apply();
    }

    super.onPause();
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.options, menu);

  return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case android.R.id.home:
      pager.setCurrentItem(0, false);
      return(true);

    case R.id.about:
      showAbout();

      return(true);

    case R.id.help:
      showHelp();

      return(true);

    case R.id.settings:
      startActivity(new Intent(this, Preferences.class));

      return(true);

    case R.id.notes:
      Intent i=new Intent(this, NoteActivity.class);

      i.putExtra(NoteActivity.EXTRA_POSITION, pager.getCurrentItem());
      startActivity(i);

      return(true);

    case R.id.update:
      WakefulIntentService.sendWakefulWork(this, DownloadCheckService.class);

      return(true);
  }

  return(super.onOptionsItemSelected(item));
}

@Override
public void onBackStackChanged() {
  if (getFragmentManager().getBackStackEntryCount() == 0) {
    LinearLayout.LayoutParams p=
        (LinearLayout.LayoutParams)sidebar.getLayoutParams();
    if (p.weight > 0) {
      p.weight=0;
      sidebar.setLayoutParams(p);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      divider.setVisibility(View.GONE);
    }
  }
}

public void onEventMainThread(BookLoadedEvent event) {
  setupPager(event.getBook());
}

private void setupPager(BookContents contents) {
  adapter=new ContentsAdapter(this, contents);
  pager.setAdapter(adapter);
  findViewById(R.id.progressBar1).setVisibility(View.GONE);
  pager.setVisibility(View.VISIBLE);

  SharedPreferences prefs=mfrag.getPrefs();
  if (prefs != null) {
    if (prefs.getBoolean(PREF_SAVE_LAST_POSITION, false)) {
      pager.setCurrentItem(prefs.getInt(PREF_LAST_POSITION, 0));
    }

    pager.setKeepScreenOn(prefs.getBoolean(PREF_KEEP_SCREEN_ON, false));
  }
}

private void openSidebar() {
  LinearLayout.LayoutParams p=
      (LinearLayout.LayoutParams)sidebar.getLayoutParams();
  if (p.weight == 0) {
    p.weight=3;
    sidebar.setLayoutParams(p);
  }
  divider.setVisibility(View.VISIBLE);
}

private void showAbout() {
  if (sidebar != null) {
    openSidebar();

    if (about == null) {
      about=SimpleContentFragment.newInstance(FILE_ABOUT);
    }

    getFragmentManager().beginTransaction().addToBackStack(null)
        .replace(R.id.sidebar, about).commit();
  }
  else {
    Intent i=new Intent(this, SimpleContentActivity.class);

    i.putExtra(SimpleContentActivity.EXTRA_FILE, FILE_ABOUT);
    startActivity(i);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private void showHelp() {
    if (sidebar != null) {
      openSidebar();

      if (help == null) {
        help=SimpleContentFragment.newInstance(FILE_HELP);
      }

      getFragmentManager().beginTransaction().addToBackStack(null)
          .replace(R.id.sidebar, help).commit();
    }
    else {
      Intent i=new Intent(this, SimpleContentActivity.class);

      i.putExtra(SimpleContentActivity.EXTRA_FILE, FILE_HELP);
      startActivity(i);
    }
  }

  private void setupStrictMode() {
    StrictMode.ThreadPolicy.Builder builder=
        new StrictMode.ThreadPolicy.Builder().detectNetwork();

    if (BuildConfig.DEBUG) {
      builder.penaltyDeath();
    }
    else {
      builder.penaltyLog();
    }

    StrictMode.setThreadPolicy(builder.build());
  }
}
```

At this point, if you build the project and run it on a `-large` or `-xlarge` device or emulator, and you choose to view the help or about pages, you will see the sidebar appear, whether in portrait or landscape.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

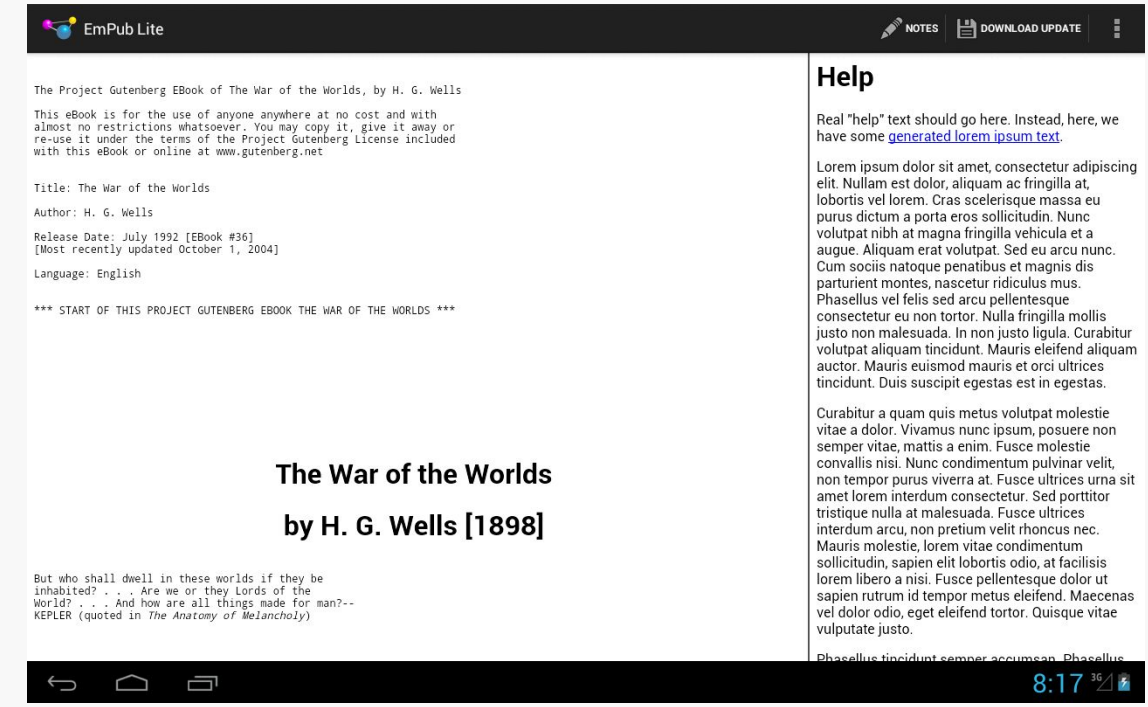


Figure 320: EmPubLite, on a Tablet-Sized Emulator, With Help

Note that a tablet emulator usually will only run acceptably fast if you are using the x86 emulator images.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Backwards Compatibility Strategies and Tactics

Android is an ever-moving target, averaging about 2.5 API level increments per year. The Android Developer site maintains a chart and table showing the most recent breakdown of OS versions making requests of the Play Store.

Most devices tend to be clustered around 1–3 minor releases. However, these are never the most recent release, which takes time to percolate through the device manufacturers and carriers and onto devices, whether those are new sales or upgrades to existing devices.

Some developers panic when they realize this.

Panic is understandable, if not necessary. This is a well-understood problem, that occurs frequently within software development — ask any Windows developer who had to simultaneously support everything from Windows 98 to Windows XP, or Windows XP through Windows 8.1. Moreover, there are many things in Android designed to make this problem as small as possible. What you need are the strategies and tactics to make it all work out.

Think Forwards, Not Backwards

Android itself tries very hard to maintain backwards compatibility. While each new Android release adds many classes and methods, relatively few are marked as deprecated, and almost none are outright eliminated. And, in Android, "deprecated" means "there's probably a better solution for what you are trying to accomplish, though we will maintain this option for you as long as we can".

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Despite this, many developers aim purely for the lowest common denominator. Aiming to support older releases is noble. Ignoring what has happened since those releases is stupid, if you are trying to distribute your app to the public via the Play Store or similar mass-distribution means.

Why? You want your app to be distinctive, not decomposing.

For example, as we saw in the chapter on the action bar, adding one line to the manifest (`android:targetSdkVersion="11"`) gives you the action bar, the holographic widget set (e.g., `Theme.Holo`), the new style of options menu, and so on. Those dead-set on avoiding things newer than Android 2.1 would not use this attribute. As a result, on Android 3.0+ devices, their apps will tend to look old. Some will not, due to other techniques they are employing (e.g., running games in a full-screen mode), but many will.

You might think that this would not matter. After all, how many people in 2011 were even using Android 3.x? 5%?

However, those in position to trumpet your application — Android enthusiast bloggers chief among them — will tend to run newer equipment. Their opinion matters, if you are trying to have their opinion sway others relative to your app. Hence, if you look out-of-touch to them, they may be less inclined to provide glowing recommendations of your app to their readers.

Besides, not everything added to newer versions of Android is pure "eye candy". It is entirely possible that features in the newer Android releases might help make your app stand out from the competition, whether it is making greater use of NFC or offering tighter integration to the stock Calendar application or whatever. By taking an "old features only" approach, you leave off these areas for improvement.

And, to top it off, the world moves faster than you think. It takes about a year for a release to go from release to majority status (or be already on the downslope towards oblivion, passed over by something newer still). You need to be careful that the decisions you make today do not doom you tomorrow. If you focus on "old features only", how much rework will it take you to catch up in six months, or a year?

Hence, this book advocates an approach that differs from that taken by many: aim high. Decide what features you want to use, whether those features are from older releases or the latest-and-greatest release. Then, write your app using those features, and take steps to ensure that everything still works reasonably well (if not as full-featured) on older devices. This too is a well-trodden path, used by Web developers

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

for ages (e.g., support sexy stuff in Firefox and Safari, while still gracefully degrading for IE6). And the techniques that those Web developers use have their analogous techniques within the Android world.

Aim Where You Are Going

One thing to bear in mind is that the OS distribution chart and table from the Android Developers blog shown above is based on requests to the Android Market.

This is only directly relevant if you are actually distributing through the Play Store.

If you are distributing through the Amazon AppStore, or to device-specific outlets (e.g., BlackBerry World), you will need to take into account what sorts of devices are using those means of distribution.

If you are specifically targeting certain non-Play Store devices, like the Kindle Fire, you will need to take into account what versions of Android they run.

If you are building an app to be distributed by a device manufacturer on a specific device, you need to know what Android version will (initially) be on that device and focus on it.

If you are distributing your app to employees of a firm, members of an organization, or the like, you need to determine if there is some specific subset of devices that they use, and aim accordingly. For example, some enterprises might distribute Android devices to their employees, in which case apps for that enterprise should run on those devices, not necessarily others.

A Target-Rich Environment

There are a few places in your application where you will need to specify Android API levels of relevance to your code.

The most important one is the `android:minSdkVersion` attribute, as discussed early in this book. You need to set this to the oldest version of Android you are willing to support, so you will not be installed on devices older than that.

There is also `android:targetSdkVersion`, mentioned in passing earlier in this chapter. In the abstract, this attribute tells Android "this is the version of Android I was thinking of when I wrote the code". Android can use this information to help

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

both backwards and forwards compatibility. Historically, this was under-utilized. However, with API Level 11 and API Level 14, `android:targetSdkVersion` took on greater importance. Specifying 11 or higher gives you the action bar and all the rest of the look-and-feel introduced in the Honeycomb release. Specifying 14 or higher will give you some new features added in Ice Cream Sandwich, such as automatic whitespace between your app widgets and other things on the user's home screen. In general, use a particular `android:targetSdkVersion` when instructions tell you to.

The third place — and perhaps the one that confuses developers the most – is the build target. This shows up as `compileSdkVersion` in `build.gradle` for Android Studio users, or in Project > Properties > Android for Eclipse users.

Part of the confusion is the multiple uses of the term "target". The build target has nothing to do with `android:targetSdkVersion`. Nor is it strictly tied to what devices you are targeting.

Rather, it is a very literal term: it is the target **of the build**. It indicates:

- What version of the Android class library you wish to compile against, dictating what classes and methods you will be able to refer to directly
- What rules to apply when interpreting resources and the manifest, to complain about things that are not recognized

The net is that you set your build target to be the lowest API level that has everything you are using directly.

Lint: It's Not Just For Belly Buttons

In the old days, the only way to find out that you were using a newer class or method than what was in your `minSdkVersion` would be to set your build target to be the same as your `minSdkVersion`. That way, any attempt to use something newer than your minimum would be greeted with compile errors. This works, but at a high cost: it makes *intentionally* using newer capabilities very painful, forcing you to use reflection to access them.

Nowadays, this is no longer needed, thanks to Lint.

Lint is part of the standard build process, adding new errors and warnings for things that are syntactically valid but probably not the right answer. In particular, Lint will

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

tell you if you are using classes or methods that are newer than your `minSdkVersion`, *even if they are valid for your build target*.

Hence, the targeting strategy nowadays is:

- Set your `minSdkVersion` to be the oldest version that you are willing to support
- Set your build target to be the version of Android that has all of the classes and methods you intend to use, allowing Lint to point out places where you need to pay attention to what sort of device you are running on (more on this later)
- Set your `targetSdkVersion` to be something relatively recent, unless you have specific reasons to use some specific version

A Little Help From Your Friends

The simplest way to use a feature yet support devices that lack the feature is to use a compatibility library that enables the feature for more devices. The Android Support package is one such compatibility library, though it also offers other classes as well.

With a compatibility library, the API for using the library is nearly identical to using the native Android capability, mostly involving slightly different package names (e.g., `android.support.v4.app.Fragment` instead of `android.app.Fragment`).

So, if there is something new that you want to use on older devices, and the new feature is not obviously tied to hardware, see if there is a "backport" of the feature available to you. Examples include backports of:

- `CalendarView` (https://github.com/SimonVT/android-calendarview)
- `Switch` (https://github.com/BoD/android-switch-backport)
- `DatePicker` (https://github.com/SimonVT/android-datepicker)
- `NumberPicker` (https://github.com/SimonVT/android-numberpicker)
- `TimePicker` (https://github.com/SimonVT/android-timepicker)

Avoid the New on the Old

If the goal is to support new capabilities on new devices, while not losing support for older devices, that implies we have the ability to determine what devices are newer and what devices are older. There are a few techniques for doing this, involving Java and resources.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Java

If you wish to conditionally execute some lines of code based on what version of Android the device is running, you can check the value of `Build.VERSION`, referring to the `android.os.Build` class. For example:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD) {
  // do something only on API Level 9 and higher
}
```

Any device running an older version of Android will skip the statements inside this version guard and therefore will not execute.

That technique is sufficient for Android 2.0 and higher devices. If you are still supporting Android 1.x devices, the story gets a bit more complicated, and that will be discussed later in the book.

If you decide that you want your build target to match your `minSdkVersion` level — as some developers elect to do — your approach will differ. Rather than *blocking* some statements from being executed on *old* devices, you will *enable* some statements to be executed on *new* devices, where those statements use Java reflection (e.g., `Class.forName()`) to reference things that are newer than what your build target supports. Since using reflection is extremely tedious in Java, it is usually simpler to have your build target reflect the classes and methods you are actually using.

@TargetAPI

One problem with this technique is that Eclipse will grumble at you, saying that you are using classes and methods not available on the API level you set for your `minSdkVersion`. To quiet down these Lint messages, you can use the `@TargetAPI` annotation.

For example, earlier in the book, we saw code like this:

```
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
static public <T> void executeAsyncTask(AsyncTask<T, ?, ?> task, T... params) {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, params);
  }
  else {
    task.execute(params);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This utility method executes an `AsyncTask` using a multi-threaded thread pool. That is the default behavior of `execute()` on API Level 10 and below. On higher versions of Android, we can explicitly opt into the multi-threaded thread pool by using `executeOnExecutor()`, but that method does not exist prior to API Level 11. Hence, we check our API level at runtime via `Build.VERSION.SDK_INT`, see if we are on `HONEYCOMB` or higher, and branch accordingly. However, for a project with a `minSdkVersion` of 10 or below, Lint will still complain — Lint is just not sophisticated enough to realize that we are correctly handling newer API levels. The `@TargetApi(Build.VERSION_CODES.HONEYCOMB)` annotation tells Lint that we have indeed confirmed that we are "doing the right thing", at least through API Level 11.

However, by using `@TargetApi(Build.VERSION_CODES.HONEYCOMB)`, we are implicitly saying that we have *not* checked to see if we are doing things properly for higher versions of Android. So long as all the classes, methods, and such that we reference in this `executeAsyncTask()` method are available in API Level 11, we are fine. If we change the implementation to reference something from, say, API Level 14, now Lint will start complaining again. **This is what we want**, so we are alerted to the problem and can fix it. Hence, *only* set the `@TargetApi()` annotation to the API level that are you explicitly handling. Do not just set it to some arbitrarily high level (or, worse, use `@SuppressWarning` to try to get Lint to shut up entirely).

Resources

The aforementioned version guards only work for Java code. Sometimes, you will want to have different resources for different versions of Android. For example, you might want to make a custom style that inherits from `Theme.Holo` for Android 3.0 and higher. Since `Theme.Holo` does not exist on earlier versions of Android, trying to use a style that inherits from it will fail miserably on, say, an Android 2.2 device.

To handle this scenario, use the `-vNN` suffix to have two resource sets. One (e.g., `res/values-v11/`) would be restricted to certain Android versions and higher (e.g., API Level 11 and higher). The default resource set (e.g., `res/values/`) would be valid for any device. However, since Android chooses more specific matches first, an Ice Cream Sandwich phone would go with the resources containing the `-v11` suffix. So, in the `-v11` resource directories, you put the resources you want used on API Level 11 and higher, and put the backwards-compatible ones in the set without the suffix. This works for Android 2.0 and higher. You can also use `-v3` for resources that *only* will be used on Android 1.5 (and no higher) or `-v4` for resources that *only* will be used on Android 1.6.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Components

One variation on the above trick allows you to conditionally enable or disable components, based on API level.

Every `<activity>`, `<receiver>`, or `<service>` in the manifest can support an `android:enabled` attribute. A disabled component (`android:enabled="false"`) cannot be started by anyone, including you.

We have already seen string resources be used in the manifest, for things like `android:label` attributes. Boolean values can also be created as resources. By convention, they are stored in a `bools.xml` file in `res/values/` or related resource sets. Just as `<string>` elements provide the definition of a string resource, `<bool>` elements provide the definition of a boolean resource. Just give the boolean resource a name and a value:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool name="on_honeycomb">false</bool>
</resources>
```

The above example has a boolean resource, named `on_honeycomb`, with a value of `false`. That would typically reside in `res/values/bools.xml`. However, you might also have a `res/values-v11/bools.xml` file, where you set `on_honeycomb` to `true`.

Now, you can use `@bool/on_honeycomb` in `android:enabled` to conditionally enable a component for API Level 11 or higher, leaving it disabled for older devices.

This can be a useful trick in cases where you might need multiple separate implementations of a component, based on API level. For example, later in the book we will examine app widgets — those interactive elements users can add to their home screens. App widgets have limited user interfaces, but API Level 11 added a few new capabilities that previously were unavailable, such as the ability to use `ListView`. However, the code for a `ListView`-backed app widget may be substantially different than for a replacement app widget that works on older devices. And, if you leave the `ListView` app widget enabled in the manifest, the user might try choosing it and crashing. So, you would only enable the `ListView` app widget on API Level 11 or higher, using the boolean resource trick.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Testing

Of course, you will want to make sure your app really does work on older devices as well as newer ones.

At build time, one trick to use periodically is to change your build target to match your `minSdkVersion`, then see where the compiler complains (or, in Eclipse, where you get all the red squiggles). If everything is known (e.g., resource attributes that will be ignored on older versions) or protected (e.g., Java statements inside a version guard `if` statement), then you are OK. If, however, you see complaints about something you forgot was only in newer Android releases, you can take steps to fix things.

You will also want to think about Android versions when it comes to testing, a topic that will be covered later in this book.

Keeping Track of Changes

Each Android SDK release is accompanied by API release notes, such as this set for Android 4.4/API Level 19.

Similarly, each Android SDK release is accompanied by its "API Differences Report", a roster of each added, removed, or modified class or method. For example, this API Differences Report points out the changes between API Level 18 and API Level 19.

Other changes are called out in the JavaDocs for `Build.VERSION_CODES`, with particular emphasis on what happens when you set a specific API level as your `android:targetSdkVersion`. Note that this roster is not complete, but may mention some things not mentioned in the other locations.

Each class, method, and field in the JavaDocs has a notation as to what API level that particular item was added. Class API levels appear towards the top of the page; method and field API levels appear on the right side of the gray bar containing the method signature or field declaration. Also, in the JavaDocs "Android APIs" column on the left, there is a drop-down that allows you to filter the contents based upon API level.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Getting Help

Obviously, this book does not cover everything. And while your #1 resource (besides the book) is going to be the Android SDK documentation, you are likely to need information beyond what's covered in either of those places.

Searching online for "android" and a class name is a good way to turn up tutorials that reference a given Android class. However, be sure to check the age of the blog post or whatever that you are reading. The older it is, the more likely that it is out of date, based upon changes in Android or just better solutions that have evolved over time.

Beyond randomly hunting around for tutorials, though, this chapter outlines some other resources to keep in mind.

Questions. Sometimes, With Answers.

The "official" places to get assistance with Android are the Android Google Groups. With respect to the SDK, there are three to consider following:

1. Stack Overflow's android tag
2. android-developers, for SDK questions and answers
3. adt-dev, for questions and answers about the official Android development tools

The author of this book also maintains the AndGlobe site, a list of Android developer support sites, with an emphasis on ones operating in languages other than English.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It is important, particularly for Stack Overflow and the Google Groups, to write well-written questions:

1. Include relevant portions of the source code (e.g., the method in which you are getting an exception) and he stack trace from LogCat, if the problem is an unhandled exception.
2. On Stack Overflow, make sure your source code and stack trace are formatted as source code; on Google Groups, consider posting long listings on gist.github.com or a similar sort of code-paste site.
3. Explain thoroughly what you are trying to do, how you are trying to do it, and why you are doing it this way (if you think your goal or approach may be a little offbeat).
4. On Stack Overflow, respond to answers and comments with your own comments, addressing the person using the @ syntax (e.g., `@CommonsWare`), to maximize the odds you will get a reply. However, only use that for people who are already involved in your question.
5. On the Google Groups, do not "ping" or reply to your own message to try to elicit a response until a reasonable amount of time has gone by (e.g., 24 hours).

Heading to the Source

The source code to Android is now available. Mostly this is for people looking to enhance, improve, or otherwise fuss with the insides of the Android operating system. But, it is possible that you will find the answers you seek in that code, particularly if you want to see how some built-in Android component "does its thing".

The source code and related resources can be found at http://source.android.com. Here, you can:

1. Download the source code
2. File bug reports against the operating system itself
3. Submit patches and learn about the process for how such patches get evaluated and approved
4. Join a separate set of Google Groups for Android platform development

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Getting Your News Fix

Ed Burnette, a nice guy who happened to write his own Android book, is also the manager of Planet Android, a feed aggregator for a number of Android-related blogs. Subscribing to the planet's feed will let you monitor quite a bit of Android-related blog posts, though not exclusively related to programming.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Code Organization and Gradle

Special Creative Commons BY-NC-SA 4.0 License Edition

Working with Library Projects

Android library projects are the primary unit of Android source reuse, particularly where that source involves more than just Java source code, such as Android resources.

In this chapter, we will explore the basics of setting up and using an Android library project.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Creating a Library Project

An Android library project, in many respects, looks like a regular Android project. It has source code and resources. It has a manifest.

What it does not do, though, is build an APK file. Instead, it represents a basket of programming assets that the Android build tools know how to blend in with regular Android projects.

How you set up an Android library project depends upon your IDE.

Android Studio

Making a project be an Android library project is simply a matter of choosing the right Gradle for Android plugin.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Rather than have:

```
apply plugin: 'com.android.application'
```

use:

```
apply plugin: 'com.android.library'
```

That's it — the `com.android.library` plugin now knows that it is creating a library, not an app.

The real question is, *where* are you making this library? In many cases, you will do so as a module in a project, where there is another module that is an app. This covers both:

- A library designed to be standalone, but with a sample app demonstrating its use
- A library designed to be used by the app, and perhaps other app modules in this project

Adding new modules to an Android Studio project is handled most simply via the new-module wizard, which you can bring up via File > New Module... from the main menu. This brings up the first page of the new-module wizard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

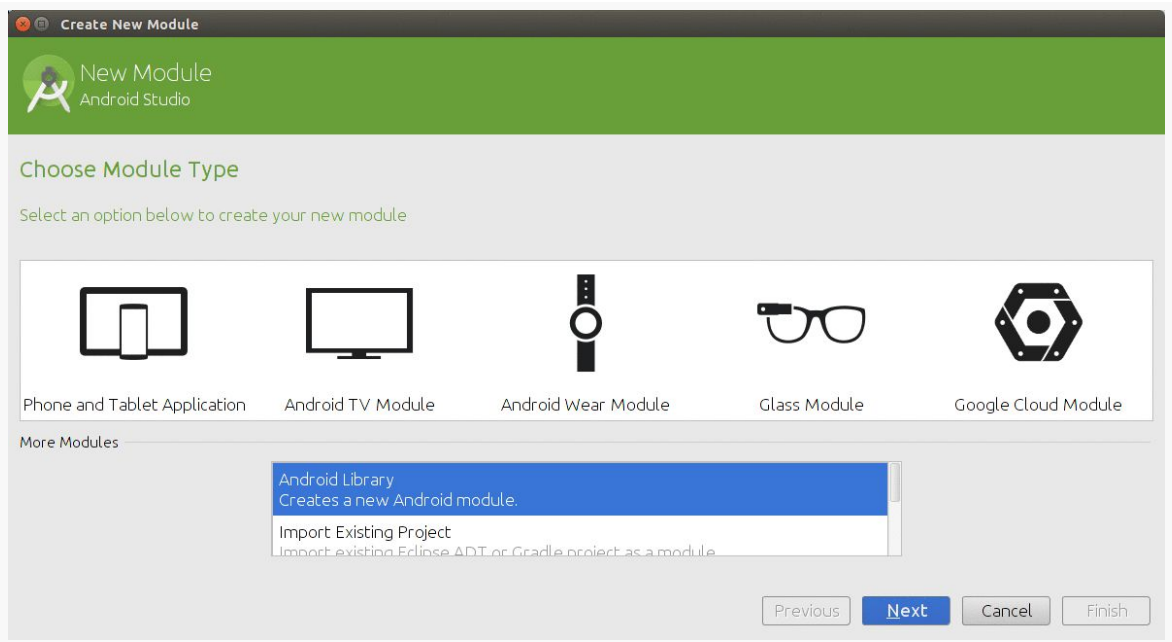


Figure 321: Android Studio New-Module Wizard, First Page

To add a library project as a module to an existing project, choose "Android Library" in the list of module types, then click Next to proceed to the second page of the wizard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

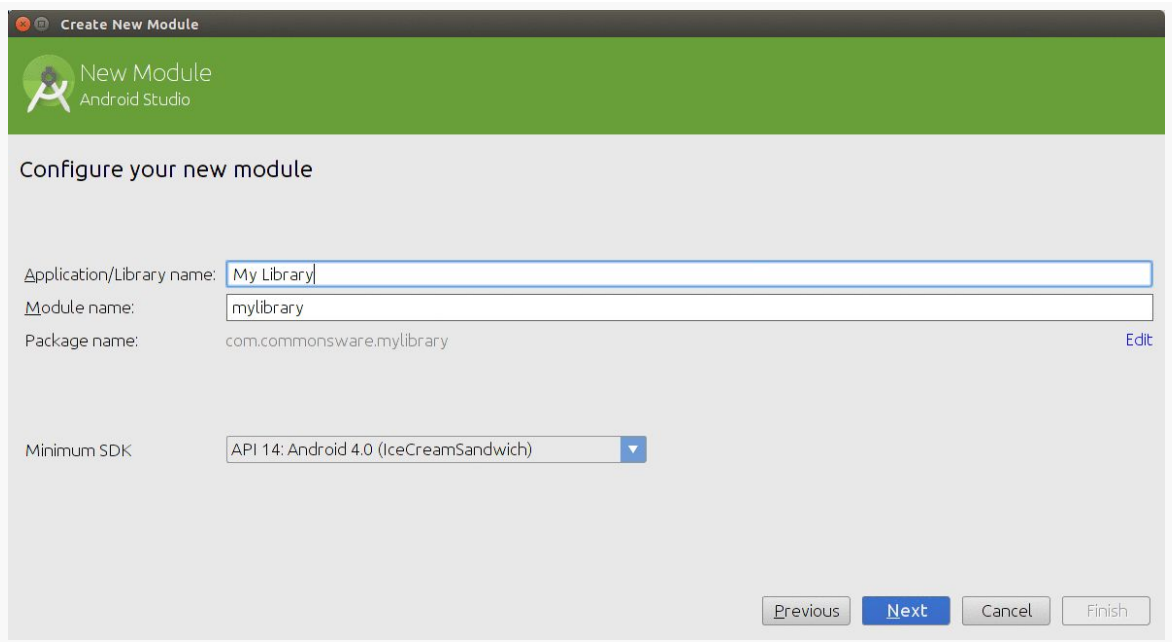


Figure 322: Android Studio New-Module Wizard, Second Page

This collects some bits of information, including:

- the "application name", whose use in this case is unclear
- the "module name", which will be the directory in this project into which this library will be created
- the "package name", which will go into the manifest of the generated library
- your `minSdkVersion`

At this point, clicking Next will take you to the same new-activity flow that you saw when creating a new project. If you want an activity to be generated for you in this library, proceed by selecting the activity template and providing the activity template configuration data. If you do not want an activity, choose "Add No Activity" in the grid of templates, then click "Finish" to create the module.

In the end, the new-module wizard will set up the new module for you, in your designated subdirectory of the project, including modifying `settings.gradle` to list this subdirectory as being a module within the project. At this point, you will be able to start using the library within the project itself.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Eclipse

To create a library project in Eclipse, start by creating a normal Android project. Then, in the project properties window (e.g., right-click on the project and choose Properties), in the Android area, check the "Is Library" checkbox. Click "Apply", and you are done.

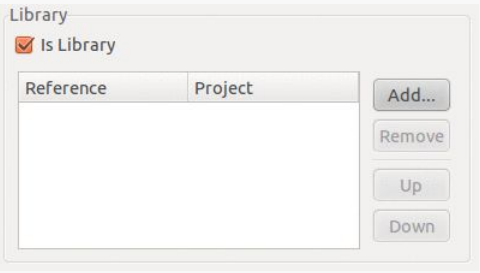


Figure 323: Android Library Project Properties, Library Section

Using a Library Project, Part II

Once you have a library project, you can attach it to a regular Android project, so the regular Android project has access to everything in the library. We covered simple scenarios for this earlier in the book. With Android Studio, you have two other major possibilities, besides what was covered previously.

If the library exists in support of a couple of your own applications, you could organize all of that into a single project with several modules (e.g., `app/` and `app2/` for the apps, with `myCoolLibrary/` for the library). How to set up this structure will be covered in the chapter on Gradle dependencies.

If the library exists in support of applications that you are not writing, such as one for another development team in your organization, or one for public distribution — you will probably wind up publishing an AAR file compiled from the library. This too is covered in the chapter on Gradle dependencies.

Library Projects and the Manifest

Library projects can publish their own `AndroidManifest.xml` file, which contributes to the overall manifest used by apps that incorporate the library. Hence, a library can:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- request permissions that perhaps are not in the app's own manifest
- publish activities or other components, without the app developer having to add entries to the app's own manifest
- stipulate a minimum SDK version required by the library code, which might be higher than the minimum SDK version required by the app itself

However, merging these manifests is a rather complex topic, and as such will be covered much later in the book.

Limitations of Library Projects

While library projects are useful for code organization and reuse, they do have their limits, such as:

- As noted above, if more than one project (main plus libraries) defines the same resource, the higher-priority project's copy gets used. Generally, that is a good thing, as it means that the main project can replace resources defined by a library (e.g., change icons). However, it does mean that two libraries might collide. It is important to keep your resource names distinct to minimize the odds of this occurrence.
- Since you are using the source code of the other project, you are subject to the limitations of its code. For example, if the third-party project is using `@Override` annotations on its implementations of interface methods, you will need to ensure that, in Eclipse, you have the compiler compliance level set to 1.6 or 1.7 — sometimes, this is set to 1.5, which complains about such annotations.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Gradle and Eclipse Projects

Projects fall into two main categories: those using the new Gradle-specific directory structure, and those that use the legacy structure that everyone used from 2008 through 2013 (and, to some extent, beyond) — mostly, projects created using Eclipse.

However, Gradle is capable of building projects in either directory layout. This chapter will review how to add Gradle support to an Eclipse Android project, without having to change your directory structure.

Prerequisites and Warnings

Understanding this chapter requires that you have read the chapter that introduces Gradle.

"Legacy"?

Here, "legacy directory structure" means a project tree that looks a bit like this:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

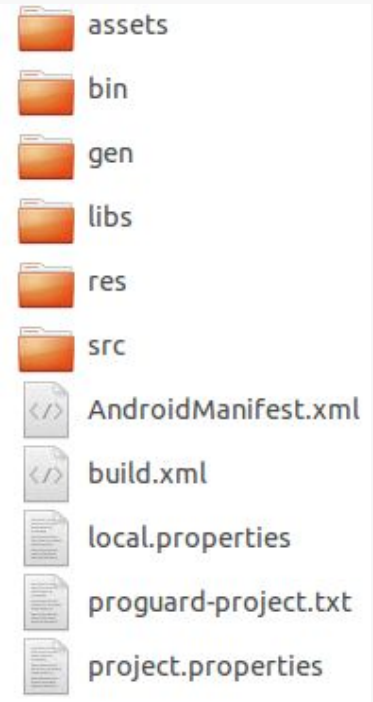


Figure 324: Legacy Directory Structure

It is dominated by a traditional Java `src/` tree, plus the Android-specific items like `res/`, `AndroidManifest.xml`, and so forth.

This directory structure will work perfectly fine with Gradle, and you may need to keep this structure for a while in order to maintain compatibility with other tools, like Eclipse.

Creating Your Gradle Build File

You need a `build.gradle` file to be able to build your project with Gradle.

As noted in the introductory chapter on Gradle, Gradle is the native build system for Android Studio. Hence, if you are using that IDE, you should get a `build.gradle` file automatically. Also, if you are moving from Eclipse to Android Studio, use the Android Studio import wizard, as it is better than your alternatives and will also help reorganize your code into the sourceset-based project structure that Android Studio (and Gradle for Android) use natively.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you are not using Android Studio, though, there are two main ways of getting a `build.gradle` file today: export one from an Eclipse project, or create one by hand. In theory, exporting from Eclipse would be the best bet. But with Eclipse being unsupported, you may wind up having to create it fully by hand. After all, as you will see, what you get from the Eclipse export process is out of date.

Exporting from Eclipse

If you have an existing Eclipse project, the easiest way to get a `build.gradle` file for that project is to let the ADT plugin export one for you.

Performing the Export

To export `build.gradle`, either choose File > Export from the Eclipse main menu, or choose "Export..." from the context menu in the Package Explorer. Either of those should bring up a wizard-style dialog where you can choose what you want to export:

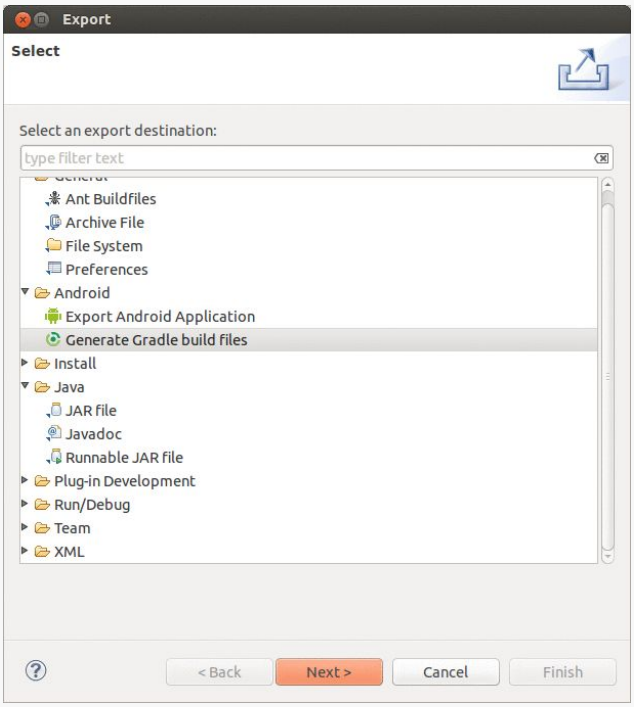


Figure 325: Eclipse Export Wizard, First Page

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here, choose "Generate Gradle build files". If that is not an option, you may be on an older version of the ADT plugin and would need to upgrade.

Clicking Next will then bring up a list of all installed projects, where you will need to check the project that you wish to export:

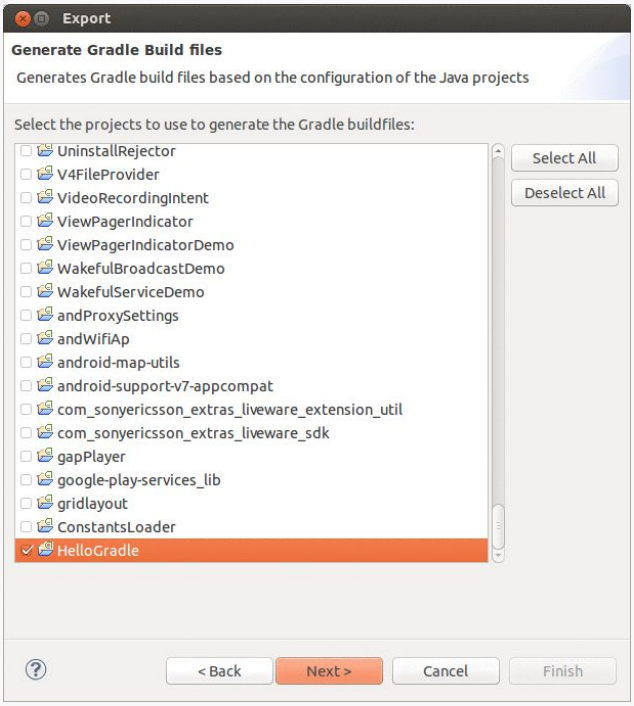


Figure 326: Eclipse Export Wizard, Second Page

Note that your project may not already be checked, due to a bug in the wizard.

Once you have checked the project, the Next button should be enabled. Clicking that will bring up a confirmation wizard page:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

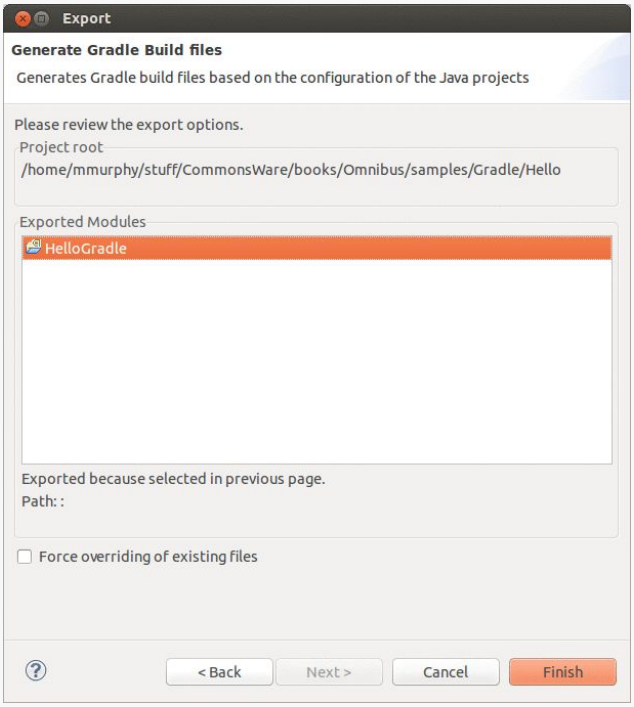


Figure 327: Eclipse Export Wizard, Third Page

It should show the project you checked in the wizard. There is a "Force overriding of existing files" checkbox — use that if you had previously exported the Gradle files and wish to replace them with a freshly-exported copy.

Clicking the Finish button will do the export and bring up a report page:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

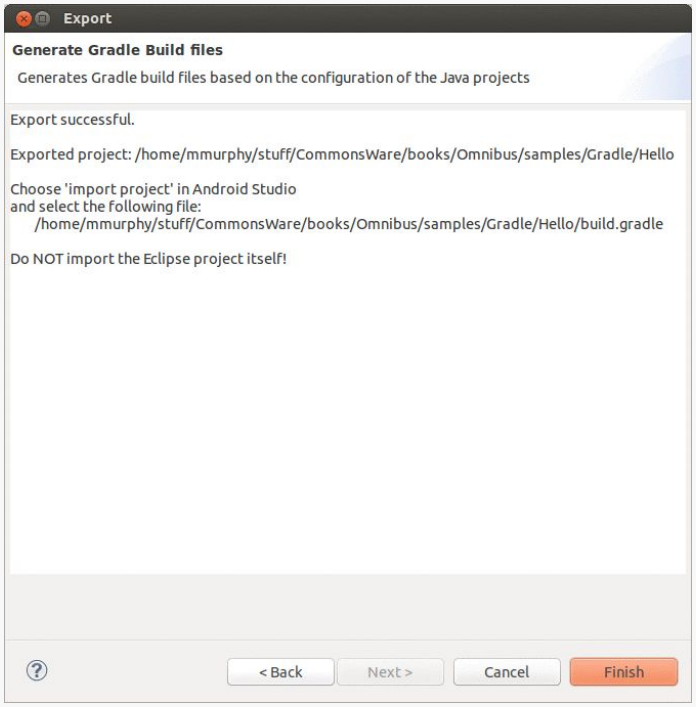


Figure 328: Eclipse Export Wizard, Fourth Page

After carefully reviewing the notes here (or possibly just ignoring them), click Finish to close the wizard.

What Gets Generated

What you get for your troubles is:

- A `build.gradle` file.
- A Gradle wrapper, in the form of a `gradlew` and/or `gradlew.bat` file and a `gradle/` subdirectory, as was discussed in the previous chapter. If you will not be using the wrapper, feel free to delete these files, except for the `gradle-wrapper.properties` file.
- A `.gradle/` hidden subdirectory, containing cached data used by the Gradle build process, such as a parsed copy of your `build.gradle` file, for faster execution if you run Gradle without having modified `build.gradle` since your last Gradle run.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What Needs Fixing

To have the resulting project work well with Android Studio, change:

- the version of the Gradle for Android plugin to `1.0.0` or higher (`classpath 'com.android.tools.build:gradle:0.12.+'` to `classpath 'com.android.tools.build:gradle:1.0.0'`)
- the `apply plugin: 'android'` statement to `apply plugin: 'com.android.application'`
- the `distributionUrl` in the `gradle/wrapper/gradle-wrapper.properties` file to `https\://services.gradle.org/distributions/gradle-2.2.1-all.zip`

Examining the Gradle File

The book's sample code contains a [Gradle/Hello] sample project. This is just a stock "Hello, world" app, as created by the Eclipse new-project wizard.

However, it also contains a `build.gradle`, exported by Eclipse:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}
apply plugin: 'com.android.application'

dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
}

android {
    compileSdkVersion 19
    buildToolsVersion "21.1.2"

    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            aidl.srcDirs = ['src']
            renderscript.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        }

        // Move the tests to tests/java, tests/res, etc...
        instrumentTest.setRoot('tests')

        // Move the build types to build-types/<type>
        // For instance, build-types/debug/java, build-types/debug/
AndroidManifest.xml, ...
        // This moves them out of them default location under src/<type>/...
which would
        // conflict with src/ being used by the main source set.
        // Adding new build types or product flavors should be accompanied
        // by a similar customization.
        debug.setRoot('build-types/debug')
        release.setRoot('build-types/release')
    }
}
```

Most of the contents of this file is covered in the introductory chapter on Gradle. The one bit that is not — the `sourcesets` closure — is covered in an upcoming chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Gradle and Tasks

A `build.gradle` file teaches Gradle how to execute tasks, such as how to compile an Android project. Outside of a Gradle-aware IDE like Android Studio, you use Gradle itself to run these tasks. If you have installed your own copy of Gradle, you would use the `gradle` command; if you are relying upon a trusted copy of the Gradle Wrapper, you would use the `./gradlew` script in your project root.

For the purposes of this book, the `gradle` command will be shown – just substitute `./gradlew` where you see `gradle` if you are using the Gradle Wrapper script.

Key Build-Related Tasks

To find out what tasks are available to you, you can run `gradle tasks` from the project directory. That will result in output akin to:

```
:tasks

------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Android tasks
-------------
androidDependencies - Displays the Android dependencies of the project
signingReport - Displays the signing info for each variant

Build tasks
-----------
assemble - Assembles all variants of all applications and secondary packages.
assembleDebug - Assembles all Debug builds
assembleDebugTest - Assembles the Test build for the Debug build
assembleRelease - Assembles all Release builds
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
clean - Deletes the build directory.

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
components - Displays the components produced by root project 'Decktastic'.
[incubating]
dependencies - Displays all dependencies declared in root project 'Decktastic'.
dependencyInsight - Displays the insight into a specific dependency in root
project 'Decktastic'.
help - Displays a help message.
projects - Displays the sub-projects of root project 'Decktastic'.
properties - Displays the properties of root project 'Decktastic'.
tasks - Displays the tasks runnable from root project 'Decktastic'.

Install tasks
-------------
installDebug - Installs the Debug build
installDebugTest - Installs the Test build for the Debug build
uninstallAll - Uninstall all applications.
uninstallDebug - Uninstalls the Debug build
uninstallDebugTest - Uninstalls the Test build for the Debug build
uninstallRelease - Uninstalls the Release build

Verification tasks
------------------
check - Runs all checks.
connectedAndroidTest - Installs and runs the tests for Build 'debug' on
connected devices.
connectedCheck - Runs all device checks on currently connected devices.
deviceCheck - Runs all device checks using Device Providers and Test Servers.
lint - Runs lint on all variants.
lintDebug - Runs lint on the Debug build
lintRelease - Runs lint on the Release build

Other tasks
-----------
compileDebugSources
compileDebugTestSources
compileReleaseSources

Rules
-----
Pattern: clean<TaskName>: Cleans the output files of a task.
Pattern: build<ConfigurationName>: Assembles the artifacts of a configuration.
Pattern: upload<ConfigurationName>: Assembles and uploads the artifacts
belonging to a configuration.
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
To see all tasks and more detail, run with --all.

BUILD SUCCESSFUL

Total time: 9.669 secs
```

This list is dynamically generated based on the contents of `build.gradle`, notably including tasks defined by the `com.android.application` plugin.

In principle, you are supposed to specify the entire task name when running that task. However, you can use shorthand, so long as it uniquely identifies the task.

Probably the most common task that a developer will use, at least in the short term, is `installDebug` (or `iD` for short). This will build a debug version of the app and install it on an available device or emulator. This roughly corresponds to `ant install debug` for those familiar with legacy Ant-based command-line builds.

Just as there is `installDebug`, there can also be `installRelease`. The `Debug` and `Release` portions of the task are not hard-coded, but rather are derived from the "build types" defined in the `build.gradle` file. The concept, role, and usage of build types will be covered in the next chapter. However, `installRelease` is not available by default, because installing an app requires that the APK be signed, and Gradle for Android does not know how to sign it. We will address this in the next chapter as well.

If you just want to build the app, without installing it, `assembleDebug` (`aD`) or `assembleRelease` (`aR`) will accomplish that aim. If you want to uninstall the app from a device or emulator, `uninstallDebug` (`uD`) and `uninstallRelease` (`uR`) should work.

Discussion of other tasks, such as the "check" tasks, will be covered in later chapters.

Results

With Eclipse, the build output went in a `bin/` directory, and generated source code (e.g., `R.java`) went in a `gen/` directory. With Gradle, all build output goes into a `build/` directory.

Specifically, your APKs will go into `build/outputs/apk`, with different APK editions based upon whether you did a debug or release build.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Gradle has a `clean` task that wipes out the `build/` directory. However, note that Eclipse does not use the `build/` directory, so a Gradle `clean` will not "clean" the pre-compiled classes and such that Eclipse uses.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Gradle and the New Project Structure

A previous chapter showed how you can use Gradle, and the Gradle for Android plugin, to do command-line builds of projects that can also work with Eclipse, IntelliJ IDEA, Ant, etc.

However, while the legacy project directory structure works, it does not let you leverage the full power of the Gradle for Android plugin. To take advantage of the build flexibility of the new build system, you will need to organize your source, resources, assets, and related files somewhat differently.

This chapter will outline this "new project structure" and show you how Gradle for Androids's concepts of build types and product flavors will make it easier for you to have multiple different forms of output from a single, albeit reorganized, project tree. This project structure is native to Android Studio, so Android Studio projects are already set up to be able to support these sorts of advanced capabilities.

NOTE: The projects demonstrated in this chapter are *not* set up to be used by Eclipse, as Eclipse does not support Gradle as of the time of this writing.

Prerequisites and Warnings

Understanding this chapter requires that you have read the chapters that introduce Gradle and cover basic Gradle/Android integration, in the context of covering the use of Gradle with the legacy project structure.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Objectives of the New Project Structure

In the beginning, Android apps tended to be pretty simple, as we only had a handful of devices, a smattering of users, one primary distribution channel (the then-Android Market) and few major investors in the Android ecosystem.

Times have changed.

Now, Android apps for public consumption can be terribly complex, let alone apps for internal enterprise use (which seem to be complex as a side effect of being developed by an enterprise). We have multiple distribution channels, such as the Amazon AppStore for Android and Yandex.Store. We have a billion devices and nearly a billion users. Brands large and small are flocking to Android, bringing with them their own challenges.

The new build system is designed to simplify creating complex Android applications, while, ideally, not making simple Android applications a lot harder. It is designed for scenarios like:

- Supporting multiple distribution channels, which may require multiple in-app purchasing engines
- Supporting one app that is customized for individual clients, such as for use by different enterprises
- Supporting an app that really needs to have different APKs for different types of devices, despite all efforts to support all devices from a single APK
- Supporting an app that is part of a much larger integrated system and needing to be built as part and parcel of that larger system
- Supporting a fleet of apps that depend upon common code, resources, third-party libraries, and the like
- And so on

The new project structure, coupled with the Gradle for Android plugin and Gradle itself, makes all of this possible... albeit with a bit of a learning curve.

Terminology

To understand what the new project structure entails, we need to define a few terms, from Gradle and the Gradle for Android plugin.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Sourcesets

To quote the Gradle documentation: "A sourceset is simply a group of source files which are compiled and executed together." Here, "source" means all the inputs that you are creating for the app, such as Java source code, Android resources and manifest files, and the like. This is in contrast to *dependencies*, which are inputs that you are (usually) obtaining from other developers, such as reusable libraries.

Sourcesets, on their own, have no particular semantic meaning. You can elect to have your project use a single sourceset, or several sourcesets, for organizing your code. You might have different sourcesets for:

- Production code versus test code, replacing the separate test project that we historically used in Android development
- Interface code versus implementation code
- Different major functional areas within the app, particularly if they are maintained by separate teams or developer pairs
- And so on

As we will see, the new project structure assumes the existence of at least one sourceset, typically named `main`, but other features of the new build system will involve additional sourcesets.

Build Types

A build type is one axis of possible alternative outcomes from the build process.

By default, the Gradle for Android plugin assumes that you want two build types: release and debug. These may go through somewhat different build steps, such as applying ProGuard to the release build but skipping it for the debug build.

With Eclipse, those two build types were "baked in" — you had them, you had *only* them, and you had limited ability to configure anything different beyond what the build tools would do differently automatically.

The Gradle for Android plugin though allows build types to have slightly different configurations, such as adding a `.debug` suffix to the APK's package name, so that you can have a release build and a debug build of your app on the same device at the same time. You also can create new build types for handling different scenarios. The new build system documentation, for example, suggests a separate "jnidebug" build

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

type, where you can indicate that the Linux `.so` files for a project should be compiled in debug mode.

As we will see, creating a new build type involves modifications to the `build.gradle` file and adding a matching sourceset.

Product Flavors

A build type is one axis for varying your output. A product flavor is another, independent axis for varying your output.

Product flavors are designed for scenarios where you want different release output for different cases. For example, you may want to have one version of your app built to use Google's in-app purchasing APIs (for distribution through the Play Store) and another version of your app built to use Amazon's in-app purchasing APIs (for distribution through the Amazon AppStore for Android). In this case, both versions of the app will be available in release form, and you may wish to have separate debug builds as well. And *most* of the code for the two versions of the app will be the same. However, you will have different code for the different distribution channels — not only does the right code have to run for the right channel, but there is no particular value in distributing the code for one channel through the other channel.

Another example would be an app that is branded and configured for different enterprise customers. You see this a lot with Web apps — the vendor sells a branded-and-configured version of the Web app to the customer, whether that app runs on vendor-supplied hardware or customer-supplied hardware. Similarly, the maker of an Android app for collecting employee timesheets might want to offer to its customers for *their* version of the timesheet app to sport the customer's logo, tie into the customer's specific accounting server, enable or disable features based upon how the customer uses timesheets, and so on. However, *most* of the code is shared between all customers, and so when the app is updated to add features or fix bugs, new builds are needed for *all* of the customers. In this case, each customer can be set up as an independent product flavor, sharing much of the code, but with slightly different code, resources (e.g., logo), and configuration based upon that customer's needs.

Product flavors are optional. If you do not describe any product flavors in your `build.gradle` file, it is assumed that you have a single product flavor, referred to internally as `default`. Many apps will not need product flavors; this is a feature that you will opt into as needed.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

As we will see, creating a new product flavor involves modifications to the `build.gradle` file and adding a matching sourceset.

Build Variants

The term "build variant" is used for the cross product of the build types and the product flavors. So, a project with `debug` and `release` build types and `google` and `amazon` product flavors will result in a total of four build variants by default:

1. `debug` + `google`
2. `debug` + `amazon`
3. `release` + `google`
4. `release` + `amazon`

Flavor Dimensions

Sometimes, even this is insufficient flexibility, such as the `google` and `amazon` scenario described earlier in this section. Or, you might need separate `free` versus `paid` editions, if you want to have an up-front fee for accessing a premium version of your app.

By default, product flavors are considered to be part of a single "flavor dimension". However, you can organize your flavors into your own separate flavor dimensions (e.g., one for free versus paid, one for distribution channel).

These then add another factor into the cross-product that determines your build variants. Suppose we have a `dist` flavor dimension, consisting of `free` and `paid` product flavors, and we have a `channel` flavor dimension, consisting of `google` and `amazon` flavors. Now, we have a total of *8* possible build variants, when we factor in the build types:

1. `debug` + `google` + `free`
2. `debug` + `amazon` + `free`
3. `release` + `google` + `free`
4. `release` + `amazon` + `free`
5. `debug` + `google` + `paid`
6. `debug` + `amazon` + `paid`
7. `release` + `google` + `paid`
8. `release` + `amazon` + `paid`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Creating a Project in the New Structure

As of the time of this writing, there are two major ways of getting a project into the new project structure: use Android Studio, or do it by hand.

As noted in the book's earliest chapters, Android Studio's native build system is Gradle with the Gradle for Android plugin. When you create a new project through that IDE, it will automatically be set up with the new project structure.

Unfortunately, at the time of this writing, there is no way to get a project organized into the new structure out of Eclipse, simply because Eclipse does not support the new structure. Similarly, there is no command-line tool, akin to `android create project` that will create a Gradle project.

Hence, in the short term, if you are not using Android Studio, and you want a project in the new structure, you will need to craft the directory tree and `build.gradle` file yourself. That could be a matter of creating them from scratch, or it could be a matter of copying a project structure from an existing source. Martin Liersch (a.k.a., "Goddchen") has published a GitHub repository with a variety of sample projects that you can use as a source of inspiration, along with the samples presented over the rest of this chapter.

What the New Project Structure Looks Like

With all that as background, let's take a look at the `Gradle/HelloNew` sample project. This project started as an Eclipse project, then had a `build.gradle` file added to it via the Eclipse export wizard. Later, though, it was reorganized to fit the new project structure.

The Directory Tree

The pre-reorganization directory tree for the project is fairly conventional, just with some added Gradle-specific files:

```
Hello
|— AndroidManifest.xml
|— assets/
|— build.gradle
|— libs/
|    |— android-support-v4.jar
|— local.properties
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
|— proguard-project.txt
|— project.properties
|— res/
|    |— drawable-hdpi/
|    |    |— ic_launcher.png
|    |— drawable-ldpi/
|    |— drawable-mdpi/
|    |    |— ic_launcher.png
|    |— drawable-xhdpi/
|    |    |— ic_launcher.png
|    |— layout/
|    |    |— activity_main.xml
|    |— menu/
|    |    |— main.xml
|    |— values/
|    |    |— dimens.xml
|    |    |— strings.xml
|    |    |— styles.xml
|    |— values-sw600dp/
|    |    |— dimens.xml
|    |— values-sw720dp-land/
|    |    |— dimens.xml
|    |— values-v11/
|    |    |— styles.xml
|    |— values-v14/
|       |— styles.xml
|— src/
   |— com/
      |— commonsware/
         |— android/
            |— gradle/
               |— hello/
                  |— MainActivity.java
```

(note: above listing includes only files of relevance for the current discussion)

The new project structure, though, is a bit different:

```
HelloNew
|— build.gradle
|— libs/
|    |— android-support-v4.jar
|— local.properties
|— proguard-project.txt
|— project.properties
|— src/
   |— main/
      |— AndroidManifest.xml
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
|— assets/
|— java/
|    |— com/
|       |— commonsware/
|          |— android/
|             |— gradle/
|                |— hello/
|                   |— MainActivity.java
|— res/
   |— drawable-hdpi/
   |    |— ic_launcher.png
   |— drawable-ldpi/
   |— drawable-mdpi/
   |    |— ic_launcher.png
   |— drawable-xhdpi/
   |    |— ic_launcher.png
   |— layout/
   |    |— activity_main.xml
   |— menu/
   |    |— main.xml
   |— values/
   |    |— dimens.xml
   |    |— strings.xml
   |    |— styles.xml
   |— values-sw600dp/
   |    |— dimens.xml
   |— values-sw720dp-land/
   |    |— dimens.xml
   |— values-v11/
   |    |— styles.xml
   |— values-v14/
      |— styles.xml
```

While the `libs/` directory is in its original spot, along with `build.gradle` and related build files, the rest has shifted.

With the new project structure, `src/` is the root of the source *sets*, not just where the source *code* goes. There is one sourceset, named `main`, in the `src/main/` directory. In there is where the `assets/` and `res/` directories go, along with the `AndroidManifest.xml` file. And, there is a `java/` directory that contains the Java source tree (what had been in the original `src/` directory).

The build.gradle File

The `build.gradle` file is much like what we profiled back in the introductory chapter on Gradle:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}

apply plugin: 'com.android.application'

dependencies {
}

android {
    compileSdkVersion 19
    buildToolsVersion "21.1.2"
}
```

We have the `buildscript` closure to describe what we need for our build tools, the `com.android.application` plugin, and details for what version of Android we are compiling against and what version of the build tools we are using.

And, as a result, we have the standard tasks, including `installDebug`.

Configuring the Stock Build Types

The `debug` and `release` build types are ready "out of the box" for your use, with a reasonable set of defaults. However, you can change those defaults and make other adjustments to how those build types work, in addition to defining your own build types. Here, we will look at the options for changing the behavior of any build type, focusing on the stock `debug` and `release` build types.

Specifically, we will examine the `Gradle/HelloConfig` sample project, which builds upon the previous sample, modifying the behavior of both `debug` and `release`.

Sourceset

Each build type can have its own associated sourceset. If you skip the directory for it, that means that the build type is not contributing changes to the `main` sourceset.

So, in the `Gradle/HelloConfig` sample project, we have a replacement version of the `strings.xml` resource, in a `debug` sourceset:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
HelloConfig
|— build.gradle
|— HelloConfig.keystore
|— libs/
|    |— android-support-v4.jar
|— local.properties
|— proguard-project.txt
|— project.properties
|— src/
   |— debug/
   |    |— res/
   |        |— values/
   |            |— strings.xml
   |— main/
      |— AndroidManifest.xml
      |— assets/
      |— java/
      |    |— com/
      |        |— commonsware/
      |            |— android/
      |                |— gradle/
      |                    |— hello/
      |                        |— MainActivity.java
      |— res/
         |— drawable-hdpi/
         |    |— ic_launcher.png
         |— drawable-ldpi/
         |— drawable-mdpi/
         |    |— ic_launcher.png
         |— drawable-xhdpi/
         |    |— ic_launcher.png
         |— layout/
         |    |— activity_main.xml
         |— menu/
         |    |— main.xml
         |— values/
         |    |— dimens.xml
         |    |— strings.xml
         |    |— styles.xml
         |— values-sw600dp/
         |    |— dimens.xml
         |— values-sw720dp-land/
         |    |— dimens.xml
         |— values-v11/
         |    |— styles.xml
         |— values-v14/
           |— styles.xml
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That `strings.xml` contains a revised version of the `app_name`, to help make it more obvious that we are running the debug version of the app:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

  <string name="app_name">HelloGradle DEBUG</string>

</resources>
```

As we will see, resources in build types' sourcesets *replace* their counterparts in the `main`. Or, a build type could add a new resource that is missing from `main`, if desired.

build.gradle Settings

We can also use the `buildTypes` closure in `build.gradle` to configure the behavior of the `debug` and/or `release` build types. In this sample project, we alter both, plus make some other changes:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}

apply plugin: 'com.android.application'

dependencies {
}

android {
    compileSdkVersion 19
    buildToolsVersion "21.1.2"

    defaultConfig {
        versionCode 2
        versionName "1.1"
        minSdkVersion 14
        targetSdkVersion 18
    }

    signingConfigs {
        release {
            storeFile file('HelloConfig.keystore')
            keyAlias 'HelloConfig'
            storePassword 'laser.yams.heady.testy'
            keyPassword 'fw.stabs.steady.wool'
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        }
    }

    buildTypes {
        debug {
          applicationIdSuffix ".d"
          versionNameSuffix "-debug"
        }

        release {
            signingConfig signingConfigs.release
        }
    }
}
```

As was noted back in the introduction to Gradle for Android, the `defaultConfig` closure allows us to change aspects of what is found in the `AndroidManifest.xml` file, replacing anything found in the actual file from the `main` sourceset.

The `buildTypes` closure is where we configure the behavior of the build types. Each build type to be configured gets its own closure inside of `buildTypes`, and in there we can override various properties.

Notable properties that we can specify for a build type include:

- `debuggable` (to override `android:debuggable` from the `<application>` element in the manifest, to indicate that the app should be considered debuggable)
- `applicationIdSuffix` (to append to the package name specified by the manifest or the `defaultConfig` `applicationId` property)
- `versionNameSuffix` (to append to the version name specified by the manifest or the `defaultConfig` `versionName` property)

(**NOTE**: `applicationIdSuffix` was formerly known as `packageNameSuffix` prior to the Gradle for Android 0.11 release)

Our `debug` build type adds suffixes to the version name and the application ID role for the package name. Note that altering the application ID *only* affects the package name as seen by Android when the app is installed and when the app is run. It does *not* affect the directory in which the `R` class is built, which uses the package name from the `AndroidManifest.xml` file. It also does not affect any of the Java packages for our own classes, which are whatever we used when we wrote them. Hence, much of our code will be oblivious to the package name change. However, if you want to reference the *real* package name, such as for looking things up in

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

PackageManager or for use with constructing a ComponentName, use getPackageName() on any Context (like an Activity), rather than some hard-coded string, as getPackageName() returns what the runtime environment thinks the package is, which will include any suffixes added during the build process. Or, use BuildConfig.APPLICATION_ID, in cases where you do not have a Context handy on which to call getPackageName().

We can also have a signingConfig property, for configuring how our APK files are digitally signed. This will be covered in a later chapter.

Order of Precedence

Properties defined for a build type, and the properties defined for the defaultConfig will override their equivalents in the AndroidManifest.xml file. However, a build type's sourceset can *also* have its own AndroidManifest.xml file. The overall order of precedence is:

- What is in build.gradle takes precedence over...
- ...what is in a build type's AndroidManifest.xml file, which takes precedence over...
- ...what is in the main AndroidManifest.xml file

However, merging manifests in general is a complex topic, with a separate chapter later in this book.

Resources from the build type's sourceset are merged into the resources from the main sourceset, and if there are collisions, the build type's resource takes precedence. The same is true for assets.

However, the behavior of Java source is slightly different. The build type's source set is still merged with the main sourceset, but if there is a collision, the result is a build error. Either the build type or the main sourceset can define any given source file, not both. So, while debug could have one version of your/package/name/Foo.java and release could have a different version of your/package/name/Foo.java, main could not *also* have your/package/name/Foo.java. Hence, if you define a class in a build type, you may need to define that class in *all* build types, so that any references from main to that class are satisfied for all build types.

One case where this would not be required would be for debug-only activities. Suppose that you wanted an activity in your app to provide diagnostic information to developers of that app regarding the state of caches and other in-memory

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

constructs. While you could get at that stuff via a debugger, that is sometimes annoying, and just tapping on a launcher icon can be easier. But you do not want, let alone need, this diagnostic activity in your release builds. To make this work, you would put the activity's Java class only in the `debug` sourceset, along with its resources and manifest entry (complete with `MAIN/LAUNCHER` `<intent-filter>`). Since the `main` sourceset does not refer to your diagnostic activity, there is no requirement for the `release` build type to have an implementation of that Java class.

Adding Build Types

Many developers will fare just fine with the `debug` and `release` build types, perhaps with some adjustments as shown above. A few developers, though, will have other scenarios that warrant new build types. Fortunately, adding a new build type is rather easy, as seen in the [Gradle/HelloBuildType] sample project, which builds upon the previous sample, adding a new build type.

As with the built-in build types, your new build types can have their own sourcesets, by adding the appropriately-named directories underneath `src/`. And, as with the built-in build types, you can configure the new build types in the `buildTypes` closure in `build.gradle`:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}

apply plugin: 'com.android.application'

dependencies {
}

android {
    compileSdkVersion 19
    buildToolsVersion "21.1.2"

    defaultConfig {
        versionCode 2
        versionName "1.1"
        minSdkVersion 14
        targetSdkVersion 18
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    signingConfigs {
        release {
            storeFile file('HelloConfig.keystore')
            keyAlias 'HelloConfig'
            storePassword 'laser.yams.heady.testy'
            keyPassword 'fw.stabs.steady.wool'
        }
    }

    buildTypes {
        debug {
          applicationIdSuffix ".d"
          versionNameSuffix "-debug"
        }

        release {
            signingConfig signingConfigs.release
        }

        mezzanine.initWith(buildTypes.release)

        mezzanine {
            applicationIdSuffix ".mezz"
            debuggable true
        }
    }
}
```

In this project, we want a third build type, named `mezzanine`, representing a "middle ground" between a regular debug build and the release build.

To tell Android about the new build type, we need to initialize one. That is handled by the `mezzanine.initWith(buildTypes.release)` statement, which initializes the new `mezzanine` build type configuration based upon the already-defined `release` build type. From there, the subsequent `mezzanine` closure can amend the properties of that build type. In this case we:

- Put a `.mezz` suffix on the package name
- Flag the project as debuggable
- Sign with the release signing key

Since the `mezzanine` build type started with the `release` build configuration, the net effect is that we have a build that is equivalent to the release build, just with the debuggable flag set (and a unique package name).

Now, we gain Gradle tasks with `Mezzanine` in the name, like `installMezzanine`, to go along with their `Debug` and `Release` counterparts.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Adding Product Flavors and Getting Build Variants

Many apps will not need product flavors, but some will. Adding a product flavor is similar, in many respects, to adding a build type, as we will see in the Gradle/HelloProductFlavors sample project, which builds upon the previous sample, adding a pair of product flavors: vanilla and chocolate.

(note: product flavors do not have to be named after actual flavors)

Each product flavor, as with each build type, can have its own sourceset. In this sample, we have `src/vanilla/` and `src/chocolate/` directories representing a source set for each product flavor:

```
HelloProductFlavors
|— build.gradle
|— HelloConfig.keystore
|— libs/
|    |— android-support-v4.jar
|— local.properties
|— proguard-project.txt
|— project.properties
|— src/
   |— chocolate/
   |    |— java/
   |       |— com/
   |          |— commonsware/
   |             |— android/
   |                |— gradle/
   |                   |— hello/
   |                      |— MainActivityOptionsStrategy.java
   |— debug/
   |    |— res/
   |       |— values/
   |          |— strings.xml
   |— main/
   |    |— AndroidManifest.xml
   |    |— assets/
   |    |— java/
   |    |    |— com/
   |    |       |— commonsware/
   |    |          |— android/
   |    |             |— gradle/
   |    |                |— hello/
   |    |                   |— MainActivity.java
   |    |— res/
   |       |— drawable-hdpi/
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
|       |    |— ic_launcher.png
|       |— drawable-ldpi/
|       |— drawable-mdpi/
|       |    |— ic_launcher.png
|       |— drawable-xhdpi/
|       |    |— ic_launcher.png
|       |— layout/
|       |    |— activity_main.xml
|       |— menu/
|       |    |— main.xml
|       |— values/
|       |    |— dimens.xml
|       |    |— strings.xml
|       |    |— styles.xml
|       |— values-sw600dp/
|       |    |— dimens.xml
|       |— values-sw720dp-land/
|       |    |— dimens.xml
|       |— values-v11/
|       |    |— styles.xml
|       |— values-v14/
|          |— styles.xml
|— vanilla/
   |— java/
      |— com/
         |— commonsware/
            |— android/
               |— gradle/
                  |— hello/
                     |— MainActivityOptionsStrategy.java
```

In the sourcesets, we have a `MainActivityOptionsStrategy` class, one implementation per product flavor. This class is referenced by a new implementation of the `MainActivity` class in the `main` sourceset, to delegate the handling of `onOptionsItemSelected()`:

```
package com.commonsware.android.gradle.hello;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.MenuItem;

public class MainActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar
    // if it is present.
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    return(MainActivityOptionsStrategy.onOptionsItemSelected(item));
  }
}
```

Since we do not have anything much to do in the menu option, each product flavor's implementation of `MainActivityOptionsStrategy` simply logs a flavor-specific message to LogCat, such as the one shown here for `vanilla`:

```
package com.commonsware.android.gradle.hello;

import android.util.Log;
import android.view.MenuItem;

public class MainActivityOptionsStrategy {
  public static boolean onOptionsItemSelected(MenuItem item) {
    Log.d("HelloProductFlavors", "vanilla!");

    return(false);
  }
}
```

To tell Gradle for Android about our product flavors, and to configure their behavior, we have a new `productFlavors` closure in the `build.gradle` file:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}

apply plugin: 'com.android.application'

dependencies {
}

android {
    compileSdkVersion 19
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    buildToolsVersion "21.1.2"

    defaultConfig {
        versionCode 2
        versionName "1.1"
        minSdkVersion 14
        targetSdkVersion 18
    }

    signingConfigs {
        release {
            storeFile file('HelloConfig.keystore')
            keyAlias 'HelloConfig'
            storePassword 'laser.yams.heady.testy'
            keyPassword 'fw.stabs.steady.wool'
        }
    }

    buildTypes {
        debug {
          applicationIdSuffix ".d"
          versionNameSuffix "-debug"
        }

        release {
            signingConfig signingConfigs.release
        }

        mezzanine.initWith(buildTypes.release)

        mezzanine {
            applicationIdSuffix ".mezz"
            debuggable true
            signingConfig signingConfigs.release
        }
    }

    productFlavors {
        vanilla {
            applicationId "com.commonsware.android.gradle.hello.vanilla"
        }

        chocolate {
            applicationId "com.commonsware.android.gradle.hello.chocolate"
        }
    }
}
```

The `defaultConfig` is implemented using the same object type as is used for product flavors. Hence, we can configure the same things on a product flavor that we can on the `defaultConfig`, such as `applicationId`, as is done in this `build.gradle` file.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In terms of order of precedence:

- Product flavors override the `main` sourceset and the `defaultConfig`
- Build types override the product flavors

So, a `debug` build of the `vanilla` product flavor will result in a package name of `com.commonsware.android.gradle.hello.vanilla.d`.

Our task names get more numerous and more complicated, to reflect the cross product of the product flavors and build types. Now, rather than `installDebug`, `installMezzanine`, and `installRelease`, we have:

- `installChocolateDebug`
- `installChocolateMezzanine`
- `installChocolateRelease`
- `installVanillaDebug`
- `installVanillaMezzanine`
- `installVanillaRelease`

Doing the Splits

Gradle for Android 0.13 introduced `splits` as a lightweight canned replacement for product flavors for two scenarios:

- Having different APK files with different NDK binaries for ARM vs. x86 (vs. anything else)
- Having different APK files with resources for a specific screen density, important for those apps that have so many graphics that they are bumping up against distribution channel limits (e.g., 50MB on the Play Store)

All you as a developer do is request that a particular split be enabled, with limited configuration. Notably, you do not have separate Gradle configuration (e.g., `applicationId`) nor sourcesets for splits. That allows splits to be processed more quickly at build time, as the build tools can make some simplifying assumptions and avoid a lot of recompiling.

Scoping Your Splits

A split, by default, will generate one APK per *possible* type of output. For example, splitting on density will give you one APK for `ldpi`, `mdpi`, `tvdpi`, `hdpi`, `xhdpi`, `xxhdpi`,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

and `xxxhdpi`. Plus, in the case of density, you also get one "universal" APK containing support for all densities by default.

That's nice... but what if you do not need separate APKs for all of those densities? For example, if you do not ship `tvdpi` resources, there is little reason to set up an APK for it separate from, say, the `hdpi` APK.

There are two basic patterns to controlling the scope of what gets built:

1. Use an `exclude` statement to start with the defaults and remove some options
2. Use a `reset()` method to wipe out the defaults, then use an `include` statement to list what you want

In other words, `exclude` implements a blacklist, and the `reset()`/`include` combination implements a whitelist. All else being equal, a whitelist is probably a better choice, so you can explicitly line it up with what you have written in your app.

Requesting NDK Splits

In your `android` closure, you can add a `splits` closure, containing an `abi` closure, which in turn sets up the APK splits by CPU architecture:

```
splits {
  abi {
    enable true
    reset()
    include 'armeabi-v7a', 'x86'
    universalApk true
  }
}
```

Here, we:

- Enable the split (`enable true`)
- Remove the default ABIs to be included (`reset()`)
- List the ABIs that we want to be included (`include 'armeabi-v7a', 'x86'`)
- Request that a "universal APK" also be created, containing all ABIs (`universalApk true`)

The latter would be useful for distribution channels that do not allow you to upload multiple APK files for different CPU architectures. This way, you can at least

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

distribute your app there, even if it takes up more disk space than you like. By default, for the CPU architectures, you do not get a "universal APK".

Requesting Density Splits

The same basic pattern can be implemented for densities:

```
splits {
  density {
    enable true
    reset()
    include 'hdpi', 'xhdpi', 'xxhdpi'
  }
}
```

Once again, we enable the split, reset the defaults, then opt into the densities that we want.

Note, though, that a "universal APK" is always generated for densities. We do not need to have `universalApk true`, and it would appear that `universalApk false` is not an option at the present time.

Revisiting the Legacy Gradle File

Given all of this, let's revisit the `build.gradle` file exported from Eclipse:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}
apply plugin: 'com.android.application'

dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
}

android {
    compileSdkVersion 19
    buildToolsVersion "21.1.2"

    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            aidl.srcDirs = ['src']
            renderscript.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }

        // Move the tests to tests/java, tests/res, etc...
        instrumentTest.setRoot('tests')

        // Move the build types to build-types/<type>
        // For instance, build-types/debug/java, build-types/debug/
AndroidManifest.xml, ...
        // This moves them out of them default location under src/<type>/...
which would
        // conflict with src/ being used by the main source set.
        // Adding new build types or product flavors should be accompanied
        // by a similar customization.
        debug.setRoot('build-types/debug')
        release.setRoot('build-types/release')
    }
}
```

The bulk of the `android`-specific configuration in the exported `build.gradle` file comes in the `sourceSets` closure inside the `android` closure. Here, we specify what should be compiled.

Gradle allows for separate locations for items presently all dumped into `src/`: Java source, AIDL interface definitions, RenderScript source files, and Java-style resources (not to be confused with Android resources). Since this build file is for a classic directory structure, though, each of those types of input files will still be found in `src/`, so the `main` `sourceSet` points Android to `src/` for each of them. In addition, `main` indicates the name of the manifest file and the location of the `res/` and `assets/` trees.

The `main` closure in the `sourceSets` closure overrides the default locations of files that comprise the sourceset. For example, in the new project structure, the Java source code goes in `java/` within the sourceset, but in the old project structure, it goes in a `src/` directory in the project root.

The `setRoot()` calls on the stock `debug` and `release` build types indicate that their sourcesets, if they exist, should be in a separate `build-types/` directory, as the normal `src/` location is being used for Java source code.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Working with the New Project Structure in Android Studio

Not surprisingly, Android Studio has a few features designed to help you work with the various sourcesets that you elect to create and use.

The Build Variants View

When we run our project, Android Studio does not prompt us for a build type or a product flavor. It just runs the project. This begs the question of how Android Studio is determining which build variant is the one to run.

This is handled by the Build Variants view, usually docked on the left side of the Android Studio IDE window:

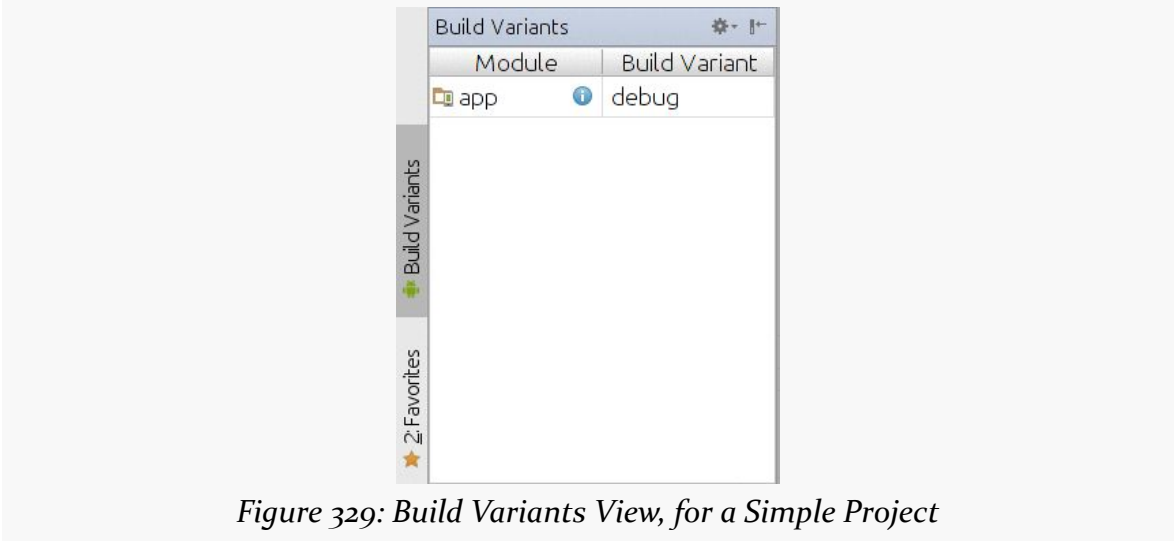


Figure 329: Build Variants View, for a Simple Project

Each of your app's modules is shown, along with the current build variant that will be used if you run that module. Tapping on the build variant will allow you to choose an alternative build variant:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

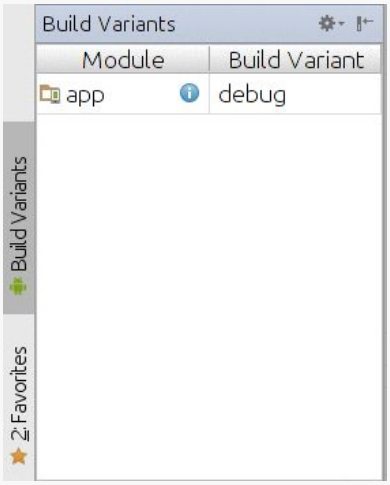


Figure 330: Build Variants View, for a Project with Custom Build Types and Product Flavors

The Android Project View

Earlier in the book, when introducing Android Studio, we saw the Android project view. Elsewhere, we saw how the Android project view can help you manage resources across multiple resource sets.

Just as the Android project view "collapses" resource set, it also collapses sourcesets:

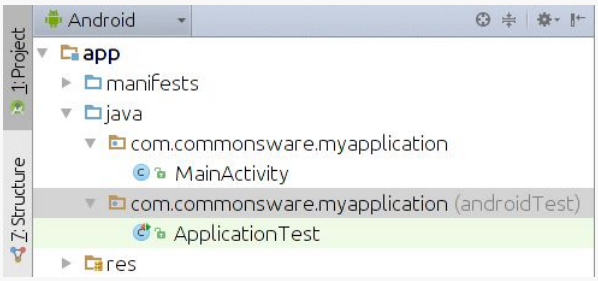


Figure 331: Android Project View, Showing Java Source

Here, we have two editions of the `com.commonsware.myapplication` package. One is just the package name, while the other has "(androidTest)" appended to it. That, as you might imagine, reflects the `main` and `androidTest` sourceset, respectively:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

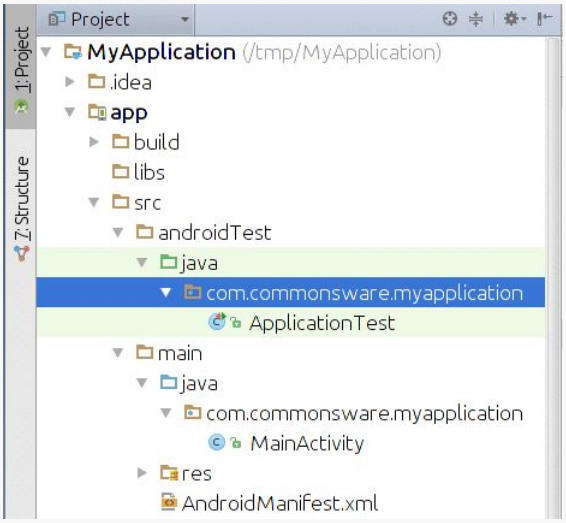


Figure 332: Classic Project View, Showing Java Source

This may be a bit useful between `main` and `androidTest`. It is likely to be far more useful if you employ product flavors, as your classes for the flavors will appear side-by-side... at least for the currently-selected flavor in the Build Variants view. At the present time, Android Studio only treats the active variant's sourcesets (plus `main` and `androidTest`) as "source", with other build variant sourcesets being considered just plain files.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Gradle and Dependencies

John Donne wrote “no man is an island”. Nowadays, few apps are islands, either. It is the rare app that can avoid using all third-party code bases. Most apps will need a backport or other class (e.g., `ViewPager`) from the Android Support package, or will rely upon the Play Services SDK, or will use any number of third party JARs and Android library projects.

The good news is that Gradle adds a lot of power for referencing these third-party code bases when you build your app. While it increases the complexity a bit for “reuse in the small” (e.g., a simple JAR), it can greatly simplify “reuse in the large” (e.g., several Android library projects).

This chapter will outline what sorts of “dependencies” your app can have and how you can configure Gradle to support them.

NOTE: The projects demonstrated in this chapter are *not* set up to be used by Eclipse, as Eclipse does not support Gradle as of the time of this writing.

Prerequisites and Warnings

Understanding this chapter requires that you have read the chapters that introduce Gradle and cover basic Gradle/Android integration, including both the legacy project structure and the new project structure.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

"Dependencies"?

In case the term is new to you, in this chapter, and in the Gradle documentation, "dependencies" means "code external to your project that your project depends upon".

In the case of Gradle-built Android apps, this includes:

- local JARs
- NDK-built local Linux `.so` files
- local Android library projects
- other types of "sub-projects"
- "artifacts" obtained from "repositories", like Maven Central

Each of these will be covered in turn in this chapter.

A Tale of Two Dependencies Closures

A `build.gradle` file — or the pair of `build.gradle` files in a classic Android Studio project — will have two `dependencies` closures.

One will be inside the `buildscript` closure, and this set of `dependencies` are dependencies for the build process itself. Here, we will list dependencies such as the Android build tools, the ones that define the `android` options we can configure.

The `dependencies` closure that is a peer of `buildscript` and `android` lists the dependencies for the project that is being build by this particular `build.gradle` file.

In other words, the `buildscript` `dependencies` are tooling dependencies, while the regular `dependencies` are compile-time sources of third-party code.

Depending Upon a JAR

Since early 2012, Ant and Eclipse shared a common rule for third-party JARs: put them in your project's `libs/` directory, and both build environments would take it from there. Specifically, they would:

- Add those JARs to the compile-time classpath, so your code that references those JARs' public APIs would compile, and

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Add the contents of those JARs to your APK, so at runtime, your references to those JARs' classes can be resolved

And it worked.

The new Gradle-based build system does not automatically use the contents of your `libs/` directory in the same way. That is why our `build.gradle` files that use simple local JARs will wind up with a `dependencies` closure that looks like this:

```
dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
}
```

The `fileTree()` will walk the directory tree rooted in the `dir` property (here, `libs`) and look for files matching the `include` wildcard pattern (here, `*.jar`). This will return all JAR files in `libs/`, which are then added to the `compile` process.

And, while we do not explicitly say to include those JARs in the resulting APK file, that is actually handled for us as part of `android` processing.

Note that this is configurable. So, if for some reason, you would prefer to have your JARs be in a directory other than `libs/`, such as `jars/` or `localDependencies/` or `wheeMakingLongDirectoryNamesIsFun/`, you are welcome to do so.

…And Why Some Do Not Like This

However, using simple JARs this way is frowned upon, at least in the absence of better options.

One reason is that a JAR file does not necessarily contain any information about the *version* of that JAR file. JARs are frequently updated, and unless the author of the JAR is "mangling in" the version information into the filename, you cannot tell by looking at a JAR whether it is old or new.

For example, the classic Android Support package's JAR is `android-support-v4.jar`. Some developers see the `-v4` part and assume that this means that this specific JAR is version 4 of the library. In reality, `-v4` means that it contains primarily the classes from `android.support.v4` and is designed for use with apps looking to support back to API Level 4. However, this JAR is updated every couple of months with new classes and bug fixes. Hence, two files named `android-support-v4.jar` may have radically different contents, if one is from 2011 and one is from 2013, for example.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This is what causes the Ant/Eclipse build process to hiccup when it encounters multiple copies of the same JAR file (e.g., one in your project and one with the same name in an Android library project). Name alone cannot distinguish whether they are the same. The build tools wound up using MD5 hashes to try to determine if the contents were indeed identical, which works but is not an ideal solution.

Another reason why people frown upon bare JARs is that there is nothing to manage the dependencies of the JAR itself. Many times a JAR depends upon other JARs, but the JAR has no way of expressing that. Instead, developers are supposed to take care of that on their own, perhaps through reading documentation and following those instructions. And, since developers do not always read documentation or follow those instructions, manual dependency management is fraught with peril.

In classic Java development, "artifacts" and "repositories" were introduced to help provide some wrapper metadata around a JAR, to help developers find the right version and determine when updates are needed. Using artifacts and repositories is recommended with Gradle, and Gradle makes it comparatively easy to use these structures, as we will see later in this chapter.

Depending Upon NDK Binaries

It is possible to use the NDK with Gradle for Android. However, this is complicated enough that it is relegated to the chapter on advanced Gradle for Android techniques.

Depending Upon an Android Library Project

Android library projects have become popular ways of sharing code between projects, as they encompass resources in addition to Java code. Some developers will use Android library projects purely internally, for reusable code between projects. Some developers will depend upon third-party library projects, such as widget libraries or Google's `appcompat-v7` backport of the action bar pattern. Some developers will publish their own libraries for third-party use.

At their core, Android library projects do not change a lot when built with Gradle as opposed to being built with Ant or Eclipse. However, the way you tell Android that a project is a library project, and how you consume such library projects, has changed substantially.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Creating a Library Project

As noted in the chapter on the Android library project, the primary difference between a regular Android project and an Android library project, in terms of Gradle configuration, is which plugin you use. Regular application projects will use the `com.android.application` plugin, while Android library projects use the `com.android.library` plugin:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}

apply plugin: 'com.android.library'

android {
    compileSdkVersion 19
    buildToolsVersion "21.1.2"
}
```

Source sets will behave the same as they do for regular apps, and otherwise your code is no different than it would be for a regular app. However, as always, an Android library project is not designed to create an APK file, but rather to serve as a library.

However, Android library projects *can* have associated test code, which is covered in the chapter on unit testing.

Depending Upon the Library Project

You have two major choices for depending upon a library project:

1. If the library project is yours, and particularly if it is only really to serve one app, you could set up the library project as a module or sub-project of the project containing that app, as we will see in the next section.
2. You can publish the library project to some repository as an AAR artifact, such as a local repository, then reference that AAR as a dependency. Coverage of artifacts and repositories appears later in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

These are not mutually exclusive. The CWAC libraries published by the author of this book, for example, use *both* techniques. These projects are published as an Android library project plus one (or more) demo app(s) demonstrating the use of the library. In `debug` builds, the demo app(s) depend upon the library as a sub-project. In `release` builds, though, the demo app(s) depend upon a published AAR of the library, to better model what other developers would see. We will see how to have dependencies vary by build type later in this chapter.

Depending Upon Sub-Projects

In classic Eclipse projects, we did not have the concept of a sub-project. However, Gradle and Android Studio both support a structure of a top-level directory (sometimes referred to as a "module") containing a series of projects.

For example, the `Gradle/HelloMultiProject` directory contains:

- a `HelloLibraryConsumer` project
- a `libraries/` subdirectory containing a `HelloLibrary` project

In this case, `HelloLibrary` is an Android library project, the one from which we saw the `build.gradle` file earlier in this chapter. `HelloLibraryConsumer` is a regular app that uses code found in the `HelloLibrary` Android library project.

If your library project is solely for use with one app, you might elect to structure your code using this approach.

The top-level directory must have a `settings.gradle` file, listing the Gradle projects found in the directory (and any of its subdirectories):

```
include ':HelloLibraryConsumer', ':libraries:HelloLibrary'
```

The leading `:` refers to the overall root directory, so `:HelloLibraryConsumer` is a reference to the `HelloLibraryConsumer/` directory under the root. Other `:` values represent levels in the hierarchy, so `:libraries:HelloLibrary` refers to the `libraries/HelloLibrary/` subdirectory. `:` is used instead of `/` as `/` is used by Gradle for other purposes.

Hence, the `include` statement tells Gradle that this aggregate project is made up of the two sub-projects.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To indicate that `HelloLibraryConsumer` wishes to consume code from the `HelloLibrary` library, another line is added to the `dependencies` closure, referencing the sub-project:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}

apply plugin: 'com.android.application'

dependencies {
    compile project(':libraries:HelloLibrary')
}

android {
    compileSdkVersion 19
    buildToolsVersion "21.1.2"
}
```

Whereas `fileTree()` is finding the JARs in the `libs/` directory, `project(':libraries:HelloLibrary')` is identifying the sub-project, and we are telling Gradle to compile that sub-project into our project.

From the overall root directory (the one with the `settings.gradle` file), you can run `gradle` tasks for the overall app. This also works from the main app's directory (in this case, `HelloLibraryConsumer/`).

While this approach is fine for some applications, this structure will not work well if the library has multiple possible consuming apps, particularly if those apps might be by other authors. In that case, the library will need to be published as an artifact to a repository, which is covered in the next section.

Depending Upon Artifacts

While JARs and sub-projects are certainly possible using Gradle for Android, the predominant approach for specifying dependencies is by referencing artifacts hosted in repositories.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What Is an Artifact?

In the context of Java-based programming, an artifact usually refers to a JAR (or other compiled output, like a Java EE WAR), accompanied by metadata that provides version information, a roster of the artifact's own dependencies, and related information.

What Is a Repository?

A repository is a collection of artifacts, stored in some location, that can be referred to in order to find and resolve requests for dependencies.

Such repositories tend to be sub-divided into "local" and "remote". A local repository is one that resides on your own development machine; a remote repository is one that resides on some server. That server could be *relatively* local (e.g., a repository in support of a corporate development team), or it could be somewhere else. Perhaps the best-known "somewhere else" is Maven Central, a repository used by many open source projects for distributing their artifacts.

Types of Artifacts and Repositories

There are two types of repositories, and associated artifact structures, supported by Gradle: Maven and Ivy. Each has their own format for the metadata and their own structure for how the files are stored.

Maven

Apache Maven is a full-fledged build system. Part of that build system is a system of artifacts and repositories. While Gradle does not use Maven's build system — rather, it largely replaces it — Gradle can consume artifacts published in a Maven-structured repository. Maven Central, as one might expect, is one such repository, but it is eminently possible to set up your own, and some organizations have done that.

As Maven seems to be the more popular of the two, this book will focus on Maven-structured repositories and Maven-style artifacts.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Ivy

Apache Ivy is an off-shoot of the Apache Ant project that gave us the Ant build system. Ivy is simply a way of declaring dependencies between components, including handling "transitive dependencies" (i.e., App A depends upon Library B, which in turn depends upon Libraries C and D).

General Artifact Dependency Setup

To depend upon artifacts, you need to teach Gradle two things:

1. Where can artifacts be found?
2. What artifact(s) do you need?

The former comes from a `repositories` closure in your `build.gradle` file, to specify the repositories that you wish to search for artifacts.

The latter comes from other variations on the `compile` statement in your `dependencies` closure. Rather than using `compile` with something like a `fileTree()`, you will specify the artifacts that you wish to use.

Artifacts are identified by three pieces of data:

1. A group
2. An artifact ID
3. A version number

These are separated by colons, so `compile 'com.commonsware.cwac:colormixer:0.5.0'` would indicate that you are seeking the artifact that:

- ...is in the `com.commonsware.cwac` group...
- ...has the `colormixer` artifact ID, and...
- ...is version `0.5.0`

Depending Upon Maven Central Artifacts

The single most common place to get artifacts is Maven Central. This is roughly analogous to the RubyGems repository for Ruby developers, or CPAN for Perl developers. Maven Central is a warehouse of many, many artifacts, only a subset of

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

which will be relevant for Android, as Maven Central has been used for many other Java environments (e.g., Java Web containers).

Your `build.gradle` file will need a `repositories` closure, at the top level (i.e., distinct from the one inside the `buildscript` closure), that requests `mavenCentral()`:

```
repositories {
    mavenCentral()
}
```

Then, you can have `compile` statements in your `dependencies` closure that list artifacts that can be found on Maven Central. One way to find out what you can include is to visit the "Gradle, please" Web site, where you can type in the name of a popular library (e.g., `otto`) and get the corresponding `dependencies` closure:

```
dependencies {
    compile 'com.squareup:otto:1.3.4'
}
```

If you have several dependencies, list them in the one `dependencies` closure, one after the next. So, for example, to depend both upon Otto and local JARs, you would have:

```
dependencies {
    compile 'com.squareup:otto:1.3.4'
    compile fileTree(dir: 'libs', include: '*.jar')
}
```

You may recall that an Android Studio project's top-level `build.gradle` file contains something like:

```
allprojects {
    repositories {
        jcenter()
    }
}
```

The `allprojects` closure represents configuration that should be applied to all modules (sub-projects) in the project. Hence, the `repositories` defined here will be defined for all modules, above and beyond any `repositories` closure in the module's own `build.gradle` file.

The `jcenter()` statement sets up Bintray's JCenter, a competitor to Maven Central.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Depending Upon Googly Artifacts

However, not all artifacts are stored at Maven Central.

One important set of artifacts stored elsewhere are Google's. Rather than have you depend upon Maven Central, they offer their own repositories, ones that you can download to your development machine via the SDK Manager. They are called "Android repository" and "Google repository", where the former is the home for things like the Android Support package, and the latter is the home for things like the Play Services SDK.

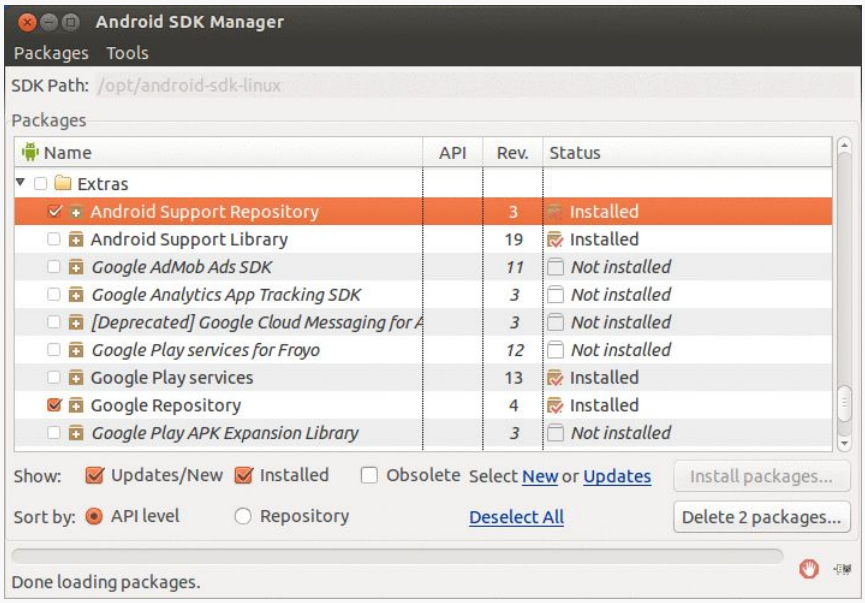


Figure 333: SDK Manager Showing Downloadable Repositories

These repositories, if found, are automatically added to your Gradle environment. So, unlike with Maven Central, you do not need to add them manually to a `repositories` closure.

If you read through the core chapters of this book, you learned about the `support-v4` and `support-v13` artifacts, providing classes like `ViewPager` and `NotificationCompat`. There are dozens of others combined between the Android Repository and Google Repository, many of which are covered elsewhere in this book.

By referencing these artifacts, you no longer need to mess around with copying JARs or attaching Android library projects to your own projects.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Depending Upon Other Artifact Repositories

Gradle supports custom artifact repositories, in Maven or Ivy style, for retrieval of artifacts. For example, your development team might have a common artifact repository for your projects, shared among the developers and a continuous integration server. Or, you may elect to publish your reusable components in your own repository, eschewing Maven Central.

The Gradle documentation covers the various possibilities, such as how to include authentication credentials for secured repositories. Typically, though, you will use a simple URL:

```
repositories {
    maven {
        url "http://repo.commonsware.com"
    }
}
```

This `repositories` closure adds a Maven-style repository, located at `http://repo.commonsware.com`.

```
repositories {
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}
```

This is an equivalent `repositories` closure, but specifies `https` as the scheme, for secure downloads of the artifacts. Since this particular repository happens to be hosted at Amazon S3, the SSL certificate requires that we use the full Amazon S3 domain name (`repo.commonsware.com.s3.amazonaws.com`) rather than the `CNAME` shorthand (`repo.commonsware.com`).

From there, you can then request to `compile` against whatever artifacts the publisher of that repository makes available.

Your Very Own Repository

You may want to have your own local repository, just on your own development machine. For example, if you are writing an Android library project, and using sub-projects to reference it is inappropriate (e.g., the library is being used by several disparate apps), you can publish your AAR artifact to your local repository, then have your other apps depend upon that artifact as found in that repository.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Consuming artifacts from your local repository is just a matter of having a `mavenLocal()` entry in your `repositories` closure:

```
repositories {
    mavenLocal()
}
```

The precise location of this repository will be platform-dependent. On Linux, for example, it is in `~/.m2/`. However, it will be on your local machine.

NOTE: On some versions of Gradle, `mavenLocal()` does not work. The workaround is:

```
repositories {
    maven { url "${System.env.HOME}/.m2/repository" } // mavenLocal()
}
```

You can have your own hosted repository, if you wish. For example, the author of this book is slowly converting his CWAC projects over to be available as JAR and AAR artifacts from the `repo.commonsware.com` repository mentioned above. From Gradle's (and Maven's) standpoint, there is no real difference between a repository hosted on a nearby file server or some remote Web server. A discussion of how to get your artifacts to such a repository is outside the scope of this book.

Publishing Libraries as Artifacts

Of course, having a local (or remote) repository is only as good as is your ability to put things into that repository. And, right now, that is a place where the current Gradle for Android plugin falls down. The documentation mentions the AAR format but offers no instructions related to publishing it, and changes in the plugin have broken many cobbled-together solutions from 2013.

The current simplest solution comes in the form of the `maven` plugin, which, as the name suggests, is a plugin for Gradle that adds support for the publishing of Maven artifacts from Gradle builds.

As this is a standard Gradle plugin, all you need to do is have `apply plugin: 'maven'` in your `build.gradle` file to use it.

Then, you can configure where and how the Maven plugin should publish your AAR, via an `uploadArchives` closure, as a top-level closure (i.e., a peer of your `android` closure):

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
apply plugin: 'maven'

uploadArchives {
    repositories.mavenDeployer {
        pom.groupId = 'com.commonsware.cwac'
        pom.artifactId = 'everything-is-awesome'
        pom.version = '0.0.1'

        repository(url: 'file:///home/somebody/put/a/real/path/in/here/kthxbye')
    }
}
```

The `groupId` and `artifactId` form the basis of the name of your artifact (in this case, a fictitious `com.commonsware.cwac:everything-is-awesome` library). The `version`, of course, is the version of the artifact. The `url` parameter on the `repository` call indicates where the artifact should be uploaded to, and this can point to a public repository (e.g., Maven Central), a private enterprise hosted repository, a local repository (as is the case here), etc.

At this point, the `gradle uploadArchives` command will build your AAR and deploy it to your designated Maven repository.

Publishing Legacy-Structured Libraries as Artifacts

Note that there is no particular requirement that your AARs be created from Android projects that use the new build system's preferred directory structure. Your AARs can come from a project that retains the legacy directory structure. This is key for the next few years, while AAR support slowly becomes dominant, so that you can support your Android library project being used in traditional source form as well.

We will see examples of using legacy structures for AAR-creating library projects later in this chapter.

About Artifact Updates

The version of the artifact that you get is determined by the version qualified in your stated dependency. There does not appear to be anything in Gradle itself to tell you about cases where there are artifacts with upgraded versions available to you. Ben Manes has published a Gradle plugin that adds a `dependencyUpdates` task that generates a report of what the status is of all of your dependencies.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Creating Android JARs from Gradle

Gradle has a long history of being used in Java development, and the standard `java` plugin for Gradle knows how to create JAR files.

However, we are not using the `java` plugin. Instead, we are using the `android` or `android-library` plugin. In the latter case, you could argue that it should support JAR-creation tasks, for libraries that do not actually use resources and so forth. Unfortunately, it does not, at least as of the time of this writing. Hence, there is no JAR-creation task available from `android` or `android-library` projects.

As is common in these cases, Jake Wharton has come to the rescue.

Jake posted an answer on a Stack Overflow question providing a quick-and-dirty bit of Gradle code to add JAR-creation tasks to an `android-library` project:

```
android.libraryVariants.all { variant ->
  def name = variant.buildType.name
  if (name.equals(com.android.builder.core.BuilderConstants.DEBUG)) {
    return; // Skip debug builds.
  }
  def task = project.tasks.create "jar${name.capitalize()}", Jar
  task.dependsOn variant.javaCompile
  task.from variant.javaCompile.destinationDir
}
```

The Gradle DSL in Groovy primarily involves building up data structures. Hence, all of our build variants wind up in a collection of objects available at `android.libraryVariants`. Jake's snippet iterates over those, tosses out those that are for debug builds, and *dynamically defines a new task*. That new task will be named `jar...`, where the `...` is the name of the build type. His snippet then configures that task to create a JAR file, after the Java code has been compiled, putting the result in the destination directory for Java compilation.

The net result is that including this snippet at the bottom of your `build.gradle` file will add tasks like `jarRelease` that will create a JAR in `build/libs/` of your project. Note that the `jarRelease` task does not appear when you run `gradle tasks`, though it will appear if you run `gradle tasks --all` to get the complete list.

This does not create a full artifact around the JAR, so if your plan was to submit this JAR to an artifact repository, you would have additional work to do. However, for the simple case of creating a JAR for manual distribution (e.g., through the "releases" area of a GitHub repository), it should work fine.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A Property of Transitive (Dependencies)

One thing to watch out for when specifying dependencies is where your dependencies' dependencies come from. Short of examining configuration files for those dependencies (e.g., their Maven POM file), you have no good way to know what your dependencies' dependencies are, let alone where they are supposed to come from.

Despite that, according to Gradleware:

> Only the repository declarations for the project whose configuration is currently resolved are taken into account, even when transitive dependencies are involved.

So, for example, suppose App A depends upon Library B, which in turn depends upon Library C. Library B is in your team's own Maven repository, while Library C comes from Maven Central. App A will need to have *both* your own Maven repository *and* Maven Central defined in the `repositories` closure, in order for Gradle to be able to obtain both libraries.

Dependencies By Build Type

A build type can have its own dependencies.

The `compile` statement in a `dependencies` closure defines dependencies for all build types. However, each build type has its own version of the `compile` statement, like `debugCompile`, that will add a dependency for use solely by that build type.

If you create your own custom build types, note that you will need to have your `dependencies` closure *after* you define the build type in the `build.gradle` file. Only after Gradle has defined your build type will your custom `compile` statement be available.

There is also `androidTestCompile`, which defines dependencies solely for use with testing. This is covered in greater detail in the chapter on unit testing.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Dependencies By Flavor

Similarly, if you define product flavors, you can have dependencies that are tied only to a particular flavor.

For example, suppose you were writing an app for various wearables, and you set up three product flavors:

```
productFlavors {
  standard {
      applicationId "com.commonsware.android.wearable.qr"
  }

  imwatch {
      applicationId "com.commonsware.android.wearable.qr.imwatch"
  }

  sony {
      applicationId "com.commonsware.android.wearable.qr.sony"
  }
}
```

A `dependencies` closure *after* the `android` closure containing the above `productFlavors` configuration could have a mix of per-flavor dependencies and flavor-specific dependencies, such as:

```
dependencies {
    compile 'com.android.support:support-v4:19.0.1'
    compile 'com.google.code.gson:gson:2.2.4'
    compile 'com.squareup.okhttp:okhttp:1.3.0'
    compile 'com.squareup.retrofit:retrofit:1.4.0'
    compile 'com.squareup.picasso:picasso:2.2.0'
    sonyCompile 'com.sonyericsson.extras.liveware.aef:SmartExtensionUtils:2.1.0'
}
```

Here, the last dependency uses `sonyCompile`, rather than `compile`, indicating that it is a dependency to be used only for the `sony` product flavor.

Note that the artifact listed for the `sonyCompile` directive does not actually exist, at least as of the time of this writing. It is possible to convert SONY's code samples into local artifacts, for reference via `mavenLocal()`, until such time as SONY starts hosting them on Maven Central or their own artifact repository.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Examining Some CWAC Builds

The author of this book publishes several open source libraries, known as the CommonsWare Android Components (CWAC). These pose an interesting challenge for conversion to Gradle, insofar as the libraries have many existing users, some of whom are still using Eclipse, Ant, or other tools based on the legacy project structure. Hence, these projects need to be able to be built sans Gradle, yet still be able to publish artifacts that can be used by Gradle.

In this section, we will examine a few of these projects, to see how the Gradle support was implemented, with a particular eye on dependencies.

A Simple CWAC Project: cwac-layouts

Most of the CWAC projects are fairly simple. Beyond having relatively few classes, most CWAC projects have no dependencies beyond Android itself. These are fairly straightforward to support with Gradle, both for building the library itself and for publishing a Gradle-compatible artifact.

For example, the CWAC-Layouts project is discussed in the chapter on custom `Views`, as it offers a few such views, particularly the mirroring classes.

The CWAC-Layouts repository has two projects: `layouts` and `demo`. The `layouts` project is the one for the library itself, while `demo` demonstrates the use of the library.

When built using Eclipse, Ant, or related tools, `layouts` is an Android library project, which `demo` depends upon via its `project.properties` file:

```
# This file is automatically generated by Android Tools.
# Do not modify this file -- YOUR CHANGES WILL BE ERASED!
#
# This file must be checked in Version Control Systems.
#
# To customize properties used by the Ant build system edit
# "ant.properties", and override values to adapt the script to your
# project structure.
#
# To enable ProGuard to shrink and obfuscate your code, uncomment this
(available properties: sdk.dir, user.home):
#proguard.config=${sdk.dir}/tools/proguard/
proguard-android.txt:proguard-project.txt

# Project target.
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
target=android-17
android.library.reference.1=../layouts
```

For the Gradle build, `demo` and `layouts` turn into modules, courtesy of `settings.gradle` in the project root directory:

```
include ':layouts', ':demo'
```

The `demo` module depends upon the `layouts` module, as it does with Eclipse/Ant builds. However, it does so in one of two ways:

- if this is a debug build, it depends on the `layouts` module, so the demo app can use the under-development version of that module
- if this is a release build, it depends upon the AAR artifact in the CommonsWare arifact repository

```
apply plugin: 'com.android.application'

repositories {
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}

dependencies {
    debugCompile project(':layouts')
    releaseCompile 'com.commonsware.cwac:layouts:0.4.+'
}

android {
    compileSdkVersion 17
    buildToolsVersion "19.1.0"

    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            aidl.srcDirs = ['src']
            renderscript.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }

        debug.setRoot('build-types/debug')
        release.setRoot('build-types/release')
    }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In all other respects, the `demo` project's `build.gradle` file is a conventional "please use the legacy project structure" implementation.

The library's `build.gradle` file is a bit more involved:

```
apply plugin: 'com.android.library'

android {
    compileSdkVersion 17
    buildToolsVersion "19.1.0"

    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            aidl.srcDirs = ['src']
            renderscript.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }

        debug.setRoot('build-types/debug')
        release.setRoot('build-types/release')
    }
}

// from http://stackoverflow.com/a/19484146/115145

android.libraryVariants.all { variant ->
  def name = variant.buildType.name
  if (name.equals(com.android.builder.core.BuilderConstants.DEBUG)) {
    return; // Skip debug builds.
  }
  def task = project.tasks.create "jar${name.capitalize()}", Jar
  task.dependsOn variant.javaCompile
  task.from variant.javaCompile.destinationDir
  task.baseName = "cwac-${PUBLISH_ARTIFACT_ID}"
  task.version = PUBLISH_VERSION
  task.exclude('com/commonsware/cwac/**/BuildConfig.**')
}

apply plugin: 'maven'

uploadArchives {
    repositories.mavenDeployer {
        pom.groupId = PUBLISH_GROUP_ID
        pom.artifactId = PUBLISH_ARTIFACT_ID
        pom.version = PUBLISH_VERSION

        repository(url: LOCAL_REPO)
    }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It uses the `maven` plugin to enable the `uploadArchives` task, as mentioned earlier in this chapter. That will compile the library project into an AAR and publish it to the development machine's local CWAC Maven repository. Separately, the author has a script that will push the necessary files to the Amazon S3-hosted CommonsWare Maven repository.

The constants referred to in the `repositories.mavenDeployer` closure come from a `gradle.properties` file, which will be covered in an upcoming chapter.

The library's `build.gradle` file also contains the custom Gradle code that adds a `jarRelease` task, as described earlier in this chapter. This task also uses values like `PUBLISH_VERSION` and `PUBLISH_ARTIFACT_ID` from the `gradle.properties` file. Hence, when the author wishes to push a new version of the library, the steps are:

1. Modify the `PUBLISH_VERSION` in the `gradle.properties` file
2. Run `gradle uploadArchives` to generate the AAR and publish it locally
3. Use an external mechanism to publish the AAR to the CommonsWare repo
4. Run `gradle jarRelease` to generate the JAR version of the project
5. Publish that JAR via the GitHub "releases" portion of the GitHub repository

CWAC-Upon-CWAC: cwac-presentation

One CWAC project that has a dependency is the CWAC-Presentation project. This is discussed in the chapter on `Presentation` and external display support, offering the `PresentationHelper` and related classes to ease the creation of apps that support external displays.

Some of those related classes use the CWAC-Layouts mirroring classes. For example, `MirrorPresentationFragment` is designed to display a mirror of a part of the primary display on an external display, such as mirroring only the slides, with the primary display also having controls for the slide presenter. Hence, CWAC-Presentation depends upon CWAC-Layouts, and that needs to be taken into account in the project build files.

As with CWAC-Layouts, the CWAC-Presentation repository has the library (`presentation`) and a demo project (`demo`). It also has a separate `demoService` project, which is set up to demonstrate another portion of the library. And, as before, for the purposes of an Ant/Eclipse build, the `demo` and `demoService` projects each have a `project.properties` file that adds a dependency upon the `presentation` library project:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
# This file is automatically generated by Android Tools.
# Do not modify this file -- YOUR CHANGES WILL BE ERASED!
#
# This file must be checked in Version Control Systems.
#
# To customize properties used by the Ant build system edit
# "ant.properties", and override values to adapt the script to your
# project structure.
#
# To enable ProGuard to shrink and obfuscate your code, uncomment this
(available properties: sdk.dir, user.home):
#proguard.config=${sdk.dir}/tools/proguard/
proguard-android.txt:proguard-project.txt

# Project target.
target=android-17
android.library.reference.1=../presentation
```

And, other than switching the dependency to be on `presentation` rather than `layouts`, the `build.gradle` file of the `demo` and `demoService` projects are the same as the one for the CWAC-Layouts:

```
apply plugin: 'com.android.application'

repositories {
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}

dependencies {
    debugCompile project(':presentation')
    releaseCompile 'com.commonsware.cwac:presentation:0.4.+'
}

android {
    compileSdkVersion 17
    buildToolsVersion "19.1.0"

    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            aidl.srcDirs = ['src']
            renderscript.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }

        debug.setRoot('build-types/debug')
        release.setRoot('build-types/release')
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
}
```

The library's `build.gradle` file is the same as the one for the CWAC-Layouts library, with two exceptions:

1. It contains a `repositories` closure, supplying the URL to the CommonsWare Maven repository
2. It adds a dependency on the CWAC-Layouts library in its `dependencies` closure

```
repositories {
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}

dependencies {
    compile 'com.commonsware.cwac:layouts:0.4.+'
}
```

The resulting AAR will have, in its Maven POM metadata file, a dependency upon CWAC-Layouts. Hence, when we build the `demo` project, it will download both the `presentation` AAR and the `layouts` AAR, to fulfill all its dependencies.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Manifest Merger Rules

When Android library projects were added as an option for app development, one problem became apparent: while libraries could contribute code and resources, they could not contribute manifest entries. Developers using libraries would sometimes have to add elements to their app manifest at the request of library authors, to add permissions, define components, and the like.

Eclipse gained some limited amount of ability to merge things from a library's manifest into a hosting app's manifest somewhere along the line, but it was all undocumented.

Gradle for Android, as of 0.10, has a much more robust and well-defined set of rules for "manifest merger". While the term "manifest merger" is still used, in reality, Gradle for Android synthesizes a manifest for your app from a variety of sources, including apps, libraries, and `build.gradle` files, also varying based upon build types and product flavors.

This chapter will help to explain a bit more about what is possible what the rules are for the manifest merger process.

Prerequisites

Understanding this chapter requires that you have read the chapters that introduce Gradle and cover basic Gradle/Android integration, including the new project structure and Gradle dependencies.

The discussion of placeholders uses Google Cloud Messaging as an example.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Manifest Scenarios

You might be wondering "why do we need all of this?" That is a fair question. Certainly, we were able to get by for quite a while without this sort of flexibility and accompanying confusion.

Here are some scenarios which help explain what you will get out of the manifest merger capabilities.

Library Manifest and App Manifest

A library — whether one of yours or one obtained via an AAR artifact from some repository — may need to augment the app's manifest. For example:

- a library for an ad network might require the `INTERNET` permission, so that apps that do not directly use the Internet still wind up requesting that permission
- a library providing a canned "about" activity might want to inject the `<activity>` element that you can use without requiring the developer to add it manually
- a library needs to be able to specify the `minSdkVersion` that it requires, which might supersede the value specified by the app, so the combined whole uses the most conservative value

App Manifest and Build Types

You may have particular needs for your main application that vary based upon build type that affect the manifest versus other things (e.g., ProGuard configuration).

For example, it may be that in debug builds, you want to have an activity that you can bring up, perhaps through `adb shell am`, that will give you diagnostic information about the app itself, or starts some diagnostic service that you can then access through your development machine's Web browser. In this case, that activity and that service would only be desired in `debug` builds, not `release` builds. And while the activity and the service code would simply be in the `debug` sourceset, you also need to merge in the manifest `<activity>` and `<service>` elements, plus perhaps other things (e.g., extra `<uses-permission>` elements that those diagnostic components need but the rest of the app does not).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

App Manifest and Product Flavors

Product flavors can override values from the `defaultConfig`, such as defining distinct `applicationId` values, and that needs to be taken into account in the combined app.

Also, product flavors might need their own manifest entries to accommodate distribution channel-specific APIs, such as swapping between Play Services and Amazon equivalents for in-app purchasing or maps.

Combo Platters

And, of course, you may have some mix of all of the above.

Pieces of Manifest Generation

When you build your app, the build tools will combine information from all of the aforementioned sources to synthesize "one true manifest" that is used for the build.

However, there may be overlaps in what the sources provide, such as both a library and the app specifying a `minSdkVersion`. Hence, there are some basic rules and control structures that you have to manage the generation process, at least somewhat.

Merger Rules

Generally speaking:

- the manifests and Gradle configurations for product flavors and build types will override...
- the app's `main/` manifest, which will override...
- the manifest of any libraries

Libraries will be considered in the order of declaration — in other words, the order that they appear in the `dependencies` closure. This includes transitive dependencies, where one dependency requires another dependency, though the exact rules here are presently unclear.

For any given element or attribute, there are specific rules for how conflicts are resolved. We will explore those later in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that values in `build.gradle`, such as the `defaultConfig` closure and its `minSdkVersion` and such, trump everything that result from the merger of disparate manifests from different sources.

Markers and Selectors

Through `tools:` attributes in the manifests that you control (i.e., not manifests from third-party libraries), you will be able to override the default rules for conflict resolution.

For example, a library might declare a particular theme to use for an activity that it publishes. That might be a reasonable default theme, but you may wish to override that theme in your app. A `tools:replace` attribute, in your `<activity>` element, will be able to teach the build tools that your `android:theme` value should replace the one from the library, whereas normally a conflict on an attribute like this would result in a build error.

You can also use "selectors" to help control in which scenarios a particular marker is applied, such as applying a marker only for a conflict arising from a specific library.

These markers and selectors will be explored in greater detail later in this chapter.

Placeholders

Sometimes, the "merger" we want to do involves something more involved.

For example, the `applicationId` and `applicationIdSuffix` properties that we set in various places in `build.gradle` can be used to allow for different variants of our builds to be installed at the same time on the same device. However, that is only true some of the time. If an app publishes a `ContentProvider`, not only does the application ID have to be unique, but so does the authority (or authorities) supported by that `ContentProvider`. This is not handled automatically, and so even though you might have the application ID distinct for different build variants, they still would conflict at install time because their provider authorities were the same.

The manifest generation process supports the notion of placeholders, where for string values in the manifest — like the `android:authorities` attribute on a `<provider>` — you can "splice in" dynamic values. One dynamic value that you can use "out of the box" is your build variant's `applicationId`, so you can have something like:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
android:authorities="${applicationId}.provider"
```

to have the authority for the provider match the build variant's `application`, but with a `.provider` suffix.

We will explore the rules around these placeholders later in this chapter.

Examining the Merger Results

The generated manifest, combining the contents of all the manifests from the app, build types, product flavors, and libraries, will wind up in the `build/intermediates/manifests/` directory of your module (e.g., `app/`). Inside that directory will be subdirectories associated with each build variant, and in those subdirectories reside the generated manifest for that build variant.

The `Manifest/Merger` sample application is designed to illustrate how these merger rules work. Note that the application does not run — it exists merely to show the results of building the APK and, along the way, generating the manifests.

This project contains an app module (`app/`) and a library module (`lib/`), with the app depending upon the library. The app module has sourcesets for both `main/` and `debug/`, the latter for debug builds. The app module also defines two product flavors, `chocolate` and `vanilla`, with a sourceset for `vanilla/`. All three sourcesets (`main/`, `debug/`, `vanilla/`) have their own `AndroidManifest.xml` files. Adding in the manifest from the library, and you have four manifests in total that may be used to create the manifest for the app. In particular, for a `vanilla debug` build, all four manifests will be relevant and merged together.

If you build the project, particularly via the `gradle` command, you will get manifests based on what builds you create. For example, `gradle assembleVanillaDebug` will create a generated manifest in `build/intermediates/manifests/vanilla/debug/`.

As you are trying to determine how manifest merging is working in your project, you may find it useful to peek at these generated manifests from time to time... as we will here in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Merging Elements and Attributes

Different sources of manifest data can contribute elements to the generated combined manifest. In many cases, these elements do not conflict, such as a library contributing a `<uses-permission>` element to an app. However, sometimes, what one source of manifest data wants is different than what another source of manifest data wants, and for that, we need to settle out what the generated manifest will contain.

Basic Merger Rules

Many element names can appear several times in a manifest, such as multiple `<uses-permission>` elements. Many of those have an identifier, usually `android:name`, that distinguishes one from the next. In general, if two manifest sources both contribute the same element (i.e., same element name, same `android:name` value), those two elements are themselves merged, which means:

- Any attributes that are in one, but not the other, are added to the combined element
- Any attributes that are in both, and are not identical in value, result in a merge conflict compile error, unless the resolution is specified via a marker
- All child elements (e.g., `<intent-filter>` inside of an `<activity>`) are merged, applying the same rules

Of course, if one manifest supplies a specific element instance, and others do not, then the specific element instance is simply included without worrying about any other merge logic.

Singleton elements — ones that could only ever appear once in the manifest — are treated as matching if they exist in more than one manifest. So, for example, the `android:versionCode` and `android:versionName` attributes of the `<manifest>` element are merged, as are attributes of `<support-screens>`, each of which can only exist once.

Example #1: Manifest Attributes

The `main/` version of the manifest defines an `android:versionName` attribute:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.merger"
  android:versionName="1.0 Main">
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  // other stuff here

</manifest>
```

None of the other manifest versions do. Hence, the `main/` version of the manifest "wins", and its `android:versionName` is used (only to perhaps be overridden by `build.gradle` values):

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.merger.vanilla"
  android:versionCode="1"
  android:versionName="1.0 Main" >

  // other stuff here

</manifest>
```

Here, we are showing the results of building the `vanilla` debug version of the app, so the package name reflects the `applicationId` defined in `build.gradle` for the `vanilla` product flavor:

```
productFlavors {
  vanilla {
    applicationId "com.commonsware.android.merger.vanilla"
  }

  chocolate {
    applicationId "com.commonsware.android.merger.chocolate"
  }
}
```

Similarly, the `versionCode` shows up because it is defined in `build.gradle`:

```
  defaultConfig {
    applicationId "com.commonsware.android.merger"
    minSdkVersion 15
    targetSdkVersion 19
    versionCode 1
  }
```

However, since `build.gradle` did not specify `versionName`, the version name comes from the manifests.

If another manifest also defined `android:versionName`, its value would need to match that of the one in `main/`, or you will get a build error from Gradle for Android... unless you use a marker, described later in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Example #2: Additional Permissions

The debug/ version of the manifest has a <uses-permission> element:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

So does the vanilla/ version, though its has android:maxSdkVersion set to 18:

```
<uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE"
  android:maxSdkVersion="18"/>
```

The manifest you get from a vanilla debug build has the android:maxSdkVersion attribute:

```
<uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE"
  android:maxSdkVersion="18" />
```

Example #3: Additional Components

The lib/ version of the manifest has an <activity>:

```
<activity android:name="ThisActivityDoesNotExist">
  <intent-filter>
    <action android:name="com.commonsware.android.merger.lib.SOMETHING_COOL" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

This is contributed by the library (or would be, if there actually was source code for the activity...). Neither the main/ nor the debug/ sourceset defines it, and so it is included verbatim in the result for chocolate builds:

```
<activity
android:name="com.commonsware.android.merger.lib.ThisActivityDoesNotExist" >
  <intent-filter>
    <action android:name="com.commonsware.android.merger.lib.SOMETHING_COOL" />

    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

Note that since the android:name attribute had a bare class name, the generated manifest expands that to include the *library's* package name (com.commonsware.android.merger.lib).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Example #4: Intent Filter

However, the `vanilla/` manifest also defines the same activity, this time with another `<intent-filter>`:

```
<activity
android:name="com.commonsware.android.merger.lib.ThisActivityDoesNotExist">
  <intent-filter>
    <action android:name="com.commonsware.android.merger.SOMETHING_VANILLA" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

Note that here, we need to have the fully-qualified class name, as we are trying to affect the library-supplied activity.

In a `vanilla` build, *both* `<intent-filter>` elements will be included by default:

```
<activity
android:name="com.commonsware.android.merger.lib.ThisActivityDoesNotExist" >
  <intent-filter>
    <action android:name="com.commonsware.android.merger.SOMETHING_VANILLA" />

    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
  <intent-filter>
    <action android:name="com.commonsware.android.merger.lib.SOMETHING_COOL" />

    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

This allows an app developer to add new ways of accessing an activity (or other component) exposed by a library.

Some Unusual Scenarios

Not everything fits the neat-and-tidy rules from the above sections and require special explanation.

uses-sdk

The `android:minSdkVersion` and `android:targetSdkVersion` from the highest-priority manifest will be used. If, however, a library's manifest specifies higher values for either of these, you will get a build error.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Hence, it is incumbent upon library authors to correctly assess how old a version of Android they are able to support, setting `android:minSdkVersion` as low as possible.

Conversely, library authors should aim to support either old or new behavior that is controlled by `android:targetSdkVersion`. For example, a library that uses `AsyncTask` should not assume that the `android:targetSdkVersion` is below 13 and therefore `execute()` will result in multi-threaded behavior on Android 3.2+. Instead, the library should use `executeOnExecutor()` on API Level 11+ devices, to specifically opt into the multi-thread thread pool, as this avoids any behavior changes based upon `android:targetSdkVersion`.

uses-feature and uses-library

The `android:required` attribute is logically OR'd among all contributors of a `<uses-feature>` element for a specific `android:name` value. In other words, if any contributor says that the feature is required, it is required. Otherwise, if one or more contributors ask for the `<uses-feature>` element but say that it is not required, it is put in the combined manifest with `android:required="false"`.

The under-utilized `<uses-library>` uses the same rule for handling the merger of its `android:required` attribute.

Markers and Selectors

Sometimes, the default merger rules will not work to your satisfaction. In particular, when there are conflicts, the build will fail, and probably that is not a desired outcome.

To declare who wins in the case of conflicts, you can use `tools:*` attributes in the manifest elements. Specifically:

- `tools:node` indicates how to resolve a conflict between two editions of this particular XML element (e.g., an `<activity>` for the same `android:name`)
- `tools:replace` indicates that certain attributes from a lower-priority edition of the manifest should be overwritten by their replacement values from a higher-priority edition of the manifest
- `tools:remove` indicates that certain attributes from a lower-priority edition of the manifest should be removed entirely

Each of these, being in the `tools` namespace, will require you to have `xmlns:tools="http://schemas.android.com/tools"` on the root `<manifest>`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

element, if it is not there already. These attributes only affect the build tools and have no runtime implications, other than in terms of how the build tools build your app based on the `tools` attributes.

For example, the main manifest has `android:uiOptions="splitActionBarWhenNarrow"` on the `<application>` element:

```
<application android:allowBackup="true"
  android:label="@string/app_name"
  android:icon="@drawable/ic_launcher"
  android:theme="@style/AppTheme"
  android:uiOptions="splitActionBarWhenNarrow">

  // other stuff here

</application>
```

The vanilla manifest wants to override this, replacing the value with `none`, so that the split action bar is not applied to the whole app. However, if the vanilla manifest just had `android:uiOptions="none"` in its `<application>` element, the build would fail, as that value conflicts with the one in the main manifest. Hence, the vanilla manifest also needs to indicate that its `android:uiOptions` value should replace the original one, via a `tools:replace` attribute:

```
<application android:allowBackup="true"
  android:label="@string/app_name"
  android:icon="@drawable/ic_launcher"
  android:theme="@style/AppTheme"
  android:uiOptions="none"
  tools:replace="android:uiOptions">

  // other stuff here

</application>
```

In the output, `android:uiOptions="none"` wins:

```
<application
  android:allowBackup="true"
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/AppTheme"
  android:uiOptions="none" >

  // other stuff here

</application>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Both `tools:replace` and `tools:remove` take a comma-delimited list of attributes that should be affected by that rule.

The `tools:node` attribute affects the entire XML element in which it resides. There are five primary values for `tools:node`:

1. `merge`, which is the default behavior described by the merger rules earlier in this chapter
2. `replace` says that the lower-priority manifest's version of this element should be replaced in its entirety with the higher-priority manifest's version of this element
3. `merge-only-attributes` says that the lower-priority manifest's version of this element should have its attributes replaced by the ones from the higher-priority manifest's version of this element, but child elements (e.g., an `<intent-filter>` underneath the annotated `<activity>` element) are left alone
4. `remove` says that the lower-priority manifest's version of this element should be removed without any replacement
5. `removeAll` says that all elements of this name (e.g., `<uses-permission>`) from lower-priority manifests should be removed, regardless of scopes like `android:name`

There is also a `strict` value that indicates that any duplication, even if it could be successfully merged, should result in a build failure. Most likely, this would be used sparingly.

By default, these `tools` attributes affect all manifests. However, it could be that you only want to affect a specific manifest, such as one coming from a certain library. In that case, `tools:selector`, in the same XML element as the other `tools:*` attributes, provides the package name of the library that the other `tools:*` attributes affect.

Employing Placeholders

The Google Cloud Messaging (GCM) system has some unusual requirements for the manifest of apps that use GCM:

- The app needs to define a custom permission, based on the application ID, via a `<permission>`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- The app needs to hold that custom permission, via a `<uses-permission>` element
- The app needs to have a `BroadcastReceiver` whose `<intent-filter>` has a `<category>` whose name is the application ID

Hence, in a GCM client app's manifest, there are three places where the application ID needs to appear. This needs to be the app's actual application ID, as may be defined either via manifests or via `applicationId` or `applicationIdSuffix` statements in a `build.gradle` file. Since the application ID can be overridden by those Gradle statements, we cannot just hard-code the application ID into the spots in the manifest.

Fortunately, part of what we get with manifest generation are placeholders.

Placeholders allow us to inject values from `build.gradle` into the manifest, particularly in XML attribute values. An `applicationId` placeholder is available automatically, and we can define custom ones via a `manifestPlaceholders` map.

For example, the main manifest for the sample project uses the `applicationId` placeholder in the requisite locations:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.android.merger"
    android:versionName="1.0 Main">

    <permission android:name="${applicationId}.C2D_MESSAGE"
        android:protectionLevel="signature" />
    <uses-permission android:name="${applicationId}.C2D_MESSAGE" />

    <application android:allowBackup="true"
        android:label="@string/app_name"
        android:icon="@drawable/ic_launcher"
        android:theme="@style/AppTheme"
        android:uiOptions="splitActionBarWhenNarrow">
      <receiver
        android:name=".GcmBroadcastReceiver"
        android:permission="com.google.android.c2dm.permission.SEND" >
        <intent-filter>
          <action android:name="com.google.android.c2dm.intent.RECEIVE" />
          <category android:name="${applicationId}" />
        </intent-filter>
      </receiver>
    </application>

</manifest>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The vanilla debug version of the generated manifest replaces those `${applicationId}` placeholders with the actual `applicationId`, such as the following for a `vanilla` build:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.merger.vanilla"
  android:versionCode="1"
  android:versionName="1.0 Main" >

  <uses-sdk
    android:minSdkVersion="15"
    android:targetSdkVersion="19" />

  <uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE"
    android:maxSdkVersion="18" />

  <permission
    android:name="com.commonsware.android.merger.vanilla.C2D_MESSAGE"
    android:protectionLevel="signature" />

  <uses-permission
android:name="com.commonsware.android.merger.vanilla.C2D_MESSAGE" />

  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme"
    android:uiOptions="none" >
    <activity
android:name="com.commonsware.android.merger.lib.ThisActivityDoesNotExist" >
      <intent-filter>
        <action android:name="com.commonsware.android.merger.SOMETHING_VANILLA"
/>

        <category android:name="android.intent.category.DEFAULT" />
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.android.merger.lib.SOMETHING_COOL"
/>

        <category android:name="android.intent.category.DEFAULT" />
      </intent-filter>
    </activity>

    <receiver
      android:name="com.commonsware.android.merger.GcmBroadcastReceiver"
      android:permission="com.google.android.c2dm.permission.SEND" >
      <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE" />
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        <category android:name="com.commonsware.android.merger.vanilla" />
      </intent-filter>
    </receiver>
  </application>

</manifest>
```

Note that the entire XML attribute value does not have to be a placeholder. For example, the `android:name` values for the `<permission>` and `<uses-permission>` elements blend the `applicationId` in with a fixed string: `android:name="${applicationId}.C2D_MESSAGE"`.

If you want additional placeholders, you can define a `manifestPlaceholders` map in `defaultConfig` or in a product flavor:

```
android {
  defaultConfig {
    manifestPlaceholders = [ foo: "bar"]
  }

  productFlavors {
    vanilla {
    }

    chocolate {
      manifestPlaceholders = [ foo: "baz" ]
    }
  }
}
```

Then, you can refer to any of your custom placeholders via the same `${}` syntax (e.g., `${foo}`, with the proper value being applied during manifest generation.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Signing Your App

Perhaps the most important step in preparing your application for production distribution is signing it with a production signing key. While mistakes here may not be immediately apparent, they can have significant long-term impacts, particularly when it comes time for you to distribute an update.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Role of Code Signing

There are many reasons why Android wants you to sign your application with a production key. Here are perhaps the top three:

- It will help distinguish your production applications from debug versions of the same applications
- Multiple applications signed with the same key can access each other's private files, if they are set up to use a shared user ID in their manifests
- You can only update an application if it has a signature from the same digital certificate

The latter one is the most important for you, if you plan on offering updates of your application. If you sign version 1.0 of your application with one key, and you sign version 2.0 of your application with another key, version 2.0 will not install over top version 1.0 — it will fail with a certificate-match error.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What Happens In Debug Mode

Of course, you may be wondering how you got this far in life without worrying about keys and certificates and signatures (unless you are using Google Maps, in which case you experienced a bit of this when you got your API key).

The Android build process creates a debug key for you automatically. That key is automatically applied when you create a debug version of your application (e.g., running the app in your IDE). This all happens behind the scenes, so it is very possible for you to go through weeks and months of development and not encounter this problem.

In fact, the most likely place where you might encounter this problem is in a distributed development environment, such as an open source project. There, you might have encountered the third bullet above, where a debug application compiled by one team member cannot install over the debug application from another team member, since they do not share a common debug key. You may have run into similar problems just on your own if you use multiple development machines (e.g., a desktop in the home office and a notebook for when you are on the road delivering Android developer training).

Finding Your Debug Keystore

The debug keystore is a `debug.keystore` file in your Android SDK data directory. This directory is not where your SDK is installed, but rather is where the tools store data unique to your account on your developer machine, such as your emulator AVDs.

This directory can be found at:

- `~/.android/` on OS X and Linux
- `C:\Documents and Settings\...\.android\` on Windows XP
- `C:\Users\...\.android\` on Windows environments newer than XP

(where `...` is your Windows username)

Synchronizing Your Debug Signing Key

If you have a development team that, for better coordination, should all use the same `debug.keystore`, just pick one and copy it to all team members' development

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

machines, replacing their generated ones. The `debug.keystore` file is a binary file and should be transferrable between operating systems (e.g., from Linux to Windows).

Production Signing Keys

Beyond the debug keystore, though, you will need one for production use. Distribution channels like the Play Store do not accept apps signed with the debug signing key. So, you will need to create a key that *is* acceptable to those channels, plus arrange to use that key when creating your production apps.

How long your production signing key is valid for is important. Once your key expires, you can no longer use it for signing new applications, which means once the key expires, you cannot update existing Android applications. Also, the Play Store requires your key to be valid beyond October 22, 2033. When you create your key, you will indicate how long it should be valid for.

Note that both the debug signing key and its production counterpart are self-signed certificates — you do not have to purchase a certificate from Verisign or anyone. These keys are for creating immutable identity, but are not for creating confirmed identity. In other words, these certificates do not prove you are such-and-so person, but can prove that the same key signed two different APKs.

Creating a Production Signing Key

The mechanics of creating a production signing key depend on whether you will use an IDE (and, if so, which one) or will create one outside of any IDE.

Android Studio

Android Studio has support to create a production signing key as part of its overall process for creating a production-signed APK, which is covered later in this chapter.

Eclipse

Eclipse also has support to create a production signing key as part of its overall process for creating a production-signed APK, which is covered later in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Manually

To manually create a production signing key, you will need to use `keytool`. This comes with the Java SDK, and so it should be available to you already.

The `keytool` utility manages the contents of a "keystore", which can contain one or more keys. Each "keystore" has a password for the store itself, and keys can also have their own individual passwords. You will need to supply these passwords later on when signing an application with the key.

Here is an example of running keytool:

```
keytool -genkey -v -keystore cw-release.keystore -alias cw-release -keyalg RSA
-validity 10000 -keysize 2048
```

The parameters used here are:

1. `-genkey`, to indicate we want to create a new key
2. `-v`, to be verbose about the key creation process
3. `-keystore`, to indicate what keystore we are manipulating (`cw-release.keystore`), which will be created if it does not already exist
4. `-alias`, to indicate what human-readable name we want to give the key (`cw-release`)
5. `-keyalg`, to indicate what public-key encryption algorithm to be using for this key (`RSA`)
6. `-validity`, to indicate how long this key should be valid, where 10,000 days or more is recommended
7. `-keysize`, for indicating the length of the signing key (2,048 bits recommended, or go higher if you prefer)

If you run the above command, you will be prompted for a number of pieces of information. If you have ever created an SSL certificate, the prompts will be familiar:

```
$ keytool -genkey -v -keystore cw-release.keystore -alias cw-release -keyalg
RSA -validity 10000 -keysize 2048
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Mark Murphy
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:  CommonsWare, LLC
What is the name of your City or Locality?
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:  PA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
correct?
  [no]:  yes

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA)
with a validity of 10,000 days
  for: CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
Enter key password for <cw-release>
  (RETURN if same as keystore password):
[Storing cw-release.keystore]
```

Signing with the Production Key

How you will apply this production signing key to sign your production app again varies by your tool chain.

Android Studio

Start by opening up your project and going to Build > Generate Signed APK from the main menu. This brings up the first page of a signing wizard:

Figure 334: Android Studio Generate Signed APK Wizard, First Page

If this is the first time you are going to sign a production app, you will need to create your production signing key, which you can do by clicking the "Create new..." button in the wizard. This brings up a separate dialog for describing the new signing key:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

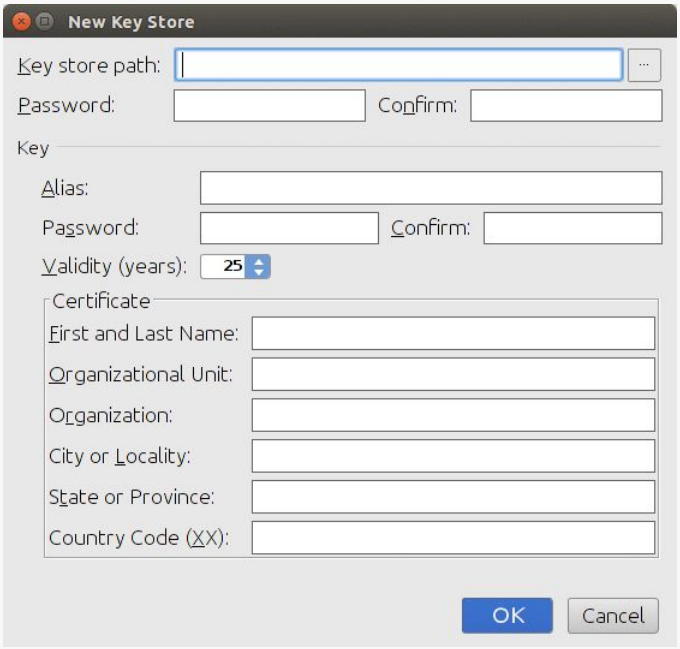


Figure 335: Android Studio New Key Store Dialog

You will need to provide a path to the keystore, manually or via the "..." button to pick a location via a dialog. You will also need to provide a password (twice) for the keystore.

You can then supply information for the signing key within the keystore, including:

- "Alias" to indicate what human-readable name we want to give the key
- "Password" and "Confirm", to specify a password for this specific key in the keystore (independent of the keystore's own password)
- "Validity", to indicate how long this key should be valid, where 25 years or more is recommended
- Details about you and your organization, asking for the standard information used in generating SSL-style keys

Clicking "OK" will generate the keystore and save it where you specified. **Be sure to back up this keystore** and record the passwords that you used.

If you already have a keystore, though, back on the first page of the "Generate Signed APK" wizard, you can click "Choose existing" to bring up a file-open dialog where you can choose your keystore. Then, fill in the keystore password, the key alias, and the key password in the dialog.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Clicking Next in the wizard brings up a page allowing you to determine what will be generated:

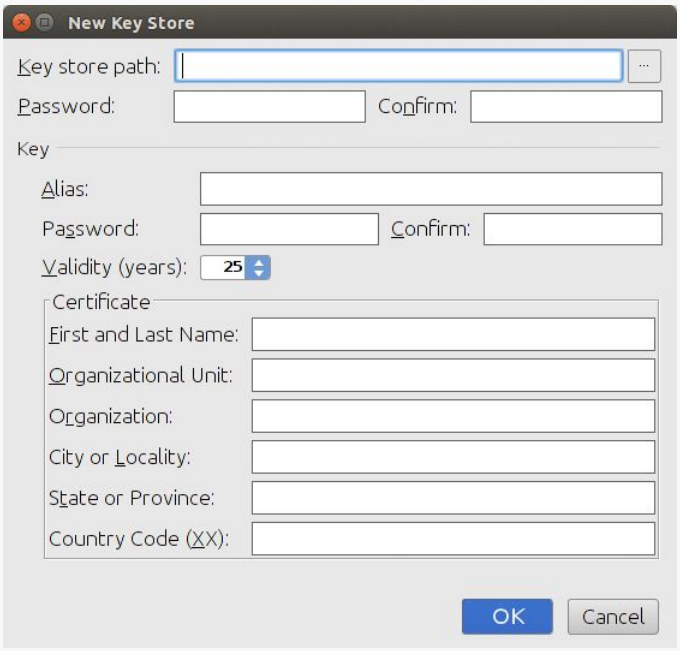


Figure 336: Android Studio Generate Signed APK Wizard, Second Page

You can indicate where the APK file should be written, what build type to use (`release` being the default), and which product flavors to use (where you can select one or several).

Clicking "Finish" will have Android Studio begin generating the APK files. This may take some time. When it is done, a dialog will appear indicating that the work is completed. In the directory that you specified, you will get one APK file per product flavor you chose, plus manifest merger reports for those APK files. And, of course, the APK files will be signed with your chosen keystore and signing key.

Gradle for Android

Gradle for Android can also be used to sign a production app. Curiously, this is completely independent of the mechanism that Android Studio uses to sign a production app. Filling in the dialogs in Android Studio does not affect your `build.gradle` file, and Android Studio's "Generate Signed APK" completely ignores any manual signing configuration that you may set up in `build.gradle` (and is discussed in this section). What is covered in this section focuses on automating

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the signing process, to be done via a build server or just running a Gradle task from the command line.

To be able to use Gradle for Android to sign your production app, you need to provide a signing configuration to the `release` build type:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}

apply plugin: 'com.android.application'

dependencies {
}

android {
    compileSdkVersion 19
    buildToolsVersion "21.1.2"

    defaultConfig {
        versionCode 2
        versionName "1.1"
        minSdkVersion 14
        targetSdkVersion 18
    }

    signingConfigs {
        release {
            storeFile file('HelloConfig.keystore')
            keyAlias 'HelloConfig'
            storePassword 'laser.yams.heady.testy'
            keyPassword 'fw.stabs.steady.wool'
        }
    }

    buildTypes {
        debug {
          applicationIdSuffix ".d"
          versionNameSuffix "-debug"
        }

        release {
            signingConfig signingConfigs.release
        }
    }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here, our `release` build type has a `signingConfig` property, referencing the name of a signing configuration specified in the `signingConfigs` closure. This is used to provide rules for how to sign the APK that is assembled by Gradle. In this project's `build.gradle` file, we have a `release` closure in `signingConfigs`, supplying the requisite information about the keystore:

- The `storeFile` path, specified as a `file()` pointing to a keystore in the project's root directory
- The `keyAlias` given to the signing key inside the keystore
- The `storePassword` and `keyPassword` used to access the keystore

The `signingConfig` property in the `release` closure in `buildTypes` references the signing configuration we want as `signingConfigs.release`. All of these Groovy closures of properties in the `build.gradle` file are effectively building up a data structure, which we can access. So, `signingConfigs.release` says to find the `release` definition in the `signingConfigs` closure.

This sample bakes in the keystore data into the `build.gradle` file, including the passwords, and has the keystore in the root of the project. That is for demonstration simplicity and will not be suitable for all projects. In particular, keystores and their credentials should *not* be stored in a publicly accessible repository, as that would allow others to sign *their* apps with *your* signing key, which is not good. There are a variety of strategies for handling this, from using environment variables to requesting the data be entered on the command line, as are discussed in the chapter on advanced Gradle techniques.

Adding a `signingConfig` property in our `release` build type enables the `installRelease` task. Running `gradle tasks` will show `installRelease` as an available option, because now Gradle for Android knows how to sign the APK. Of course, there could be flaws in the signing configuration (e.g., mis-entered key alias), and that will result in build errors when you try to `installRelease` the project.

Eclipse

To create a production signed version of your app, right-click over the app in the Package Explorer and choose Export from the context menu. In the list of things to export, choose "Export Android Application":

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

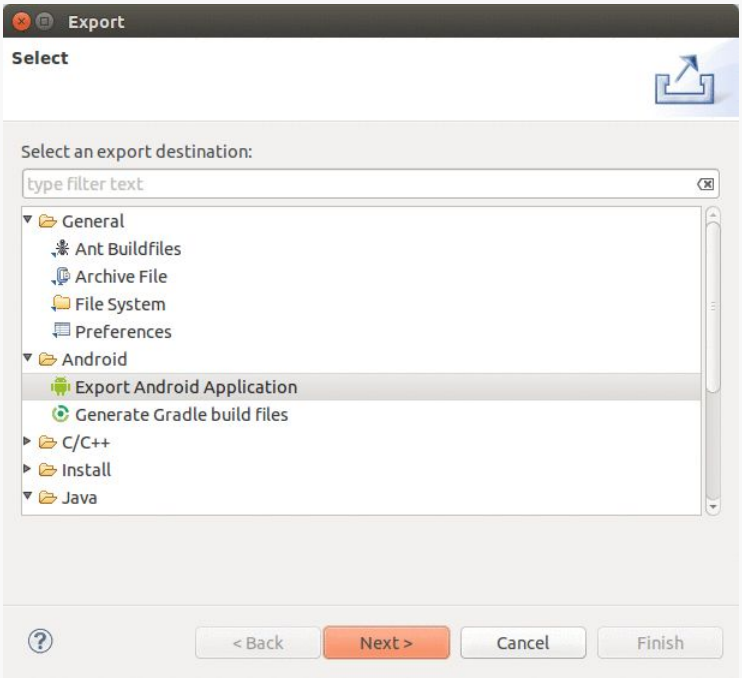


Figure 337: Eclipse Export Options

This kicks off a wizard where you can create the production-signed app. The first page of this wizard just confirms what you wish to export and whether or not there are any problems that would prevent the export, such as compile errors:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

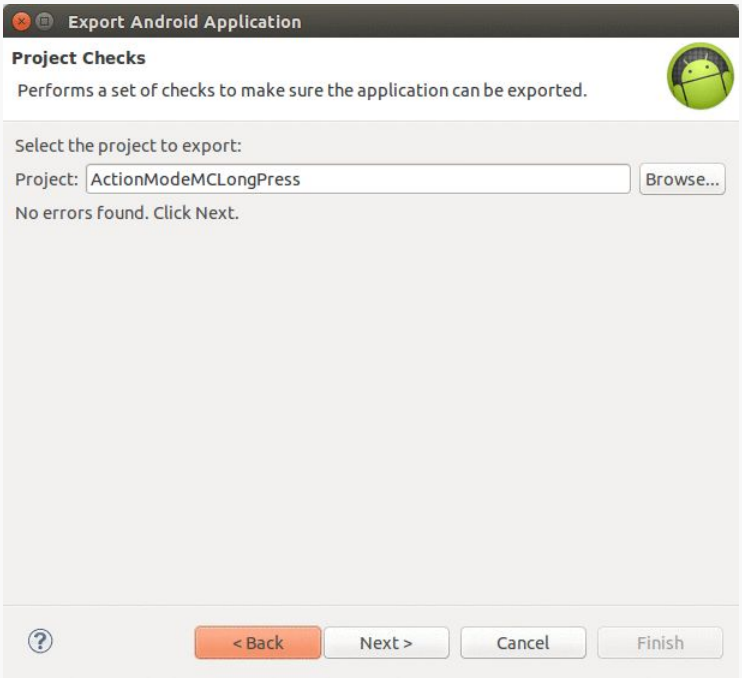


Figure 338: Eclipse Production App Wizard, First Page

Clicking the Next button brings up the second wizard page, focusing on the production keystore:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

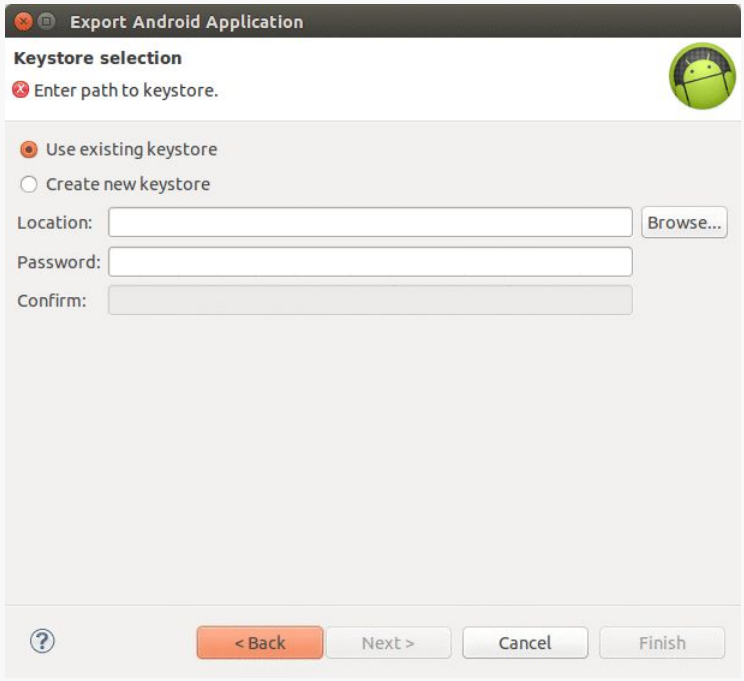


Figure 339: Eclipse Production App Wizard, Second Page

If you used this wizard previously, and you already have a production keystore, choose it via the "Browse..." button and fill in your keystore's password. If you need to create a new keystore, choose the "Create new keystore" radio button, choose where to store the keystore via the "Browse..." button, and fill in your desired keystore password.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

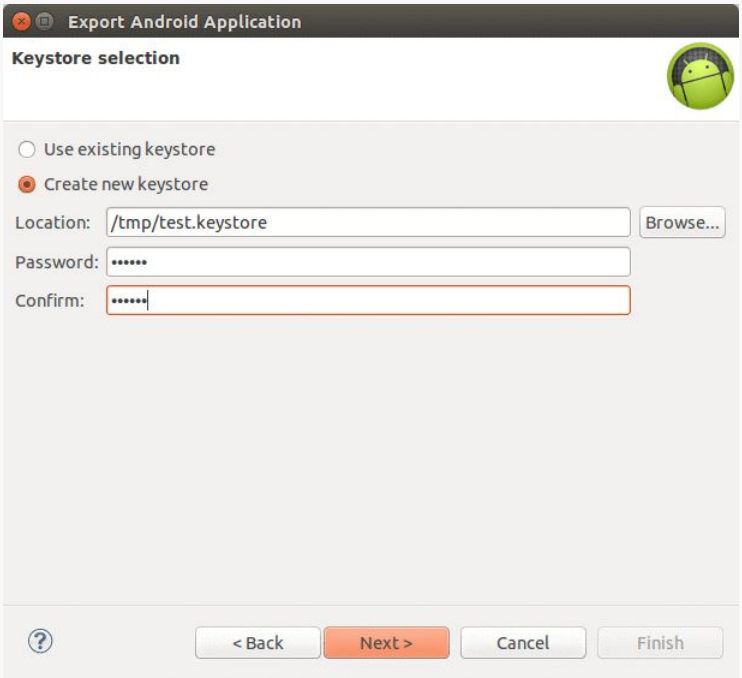


Figure 340: Eclipse Production App Wizard, Second Page, Filled In for New Keystore

If you chose to create a new keystore, the Next button brings up a dialog where you can fill in details of the production signing key:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

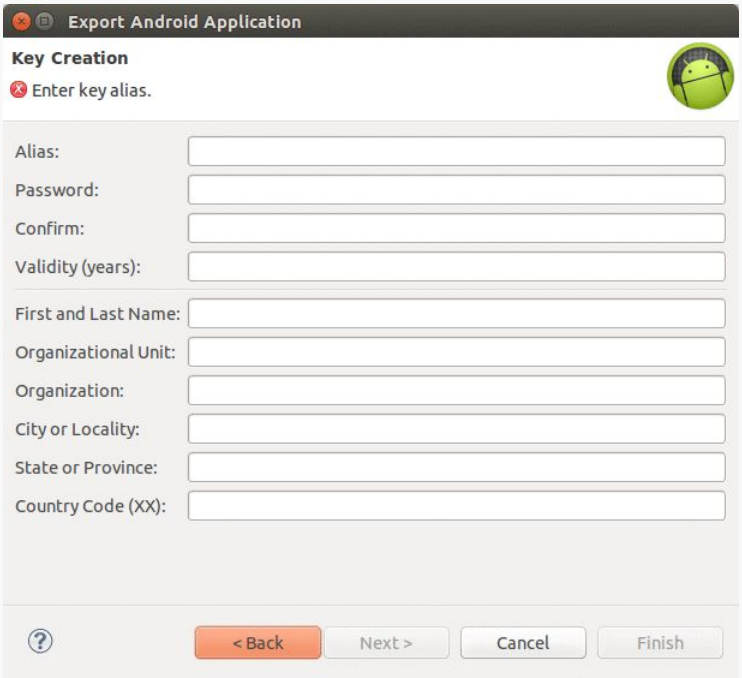


Figure 341: Eclipse Production App Wizard, New-Keystore Page

This includes:

- "Alias" to indicate what human-readable name we want to give the key
- "Password" and "Confirm", to specify a password for this specific key in the keystore (independent of the keystore's own password)
- "Validity", to indicate how long this key should be valid, where 25 years or more is recommended
- Details about you and your organization, asking for the standard information used in generating SSL-style keys

If, instead, you chose to open an existing keystore, the Next button would take you to a wizard page to choose your desired signing key and provide *its* password, or to create a new signing key within the keystore:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

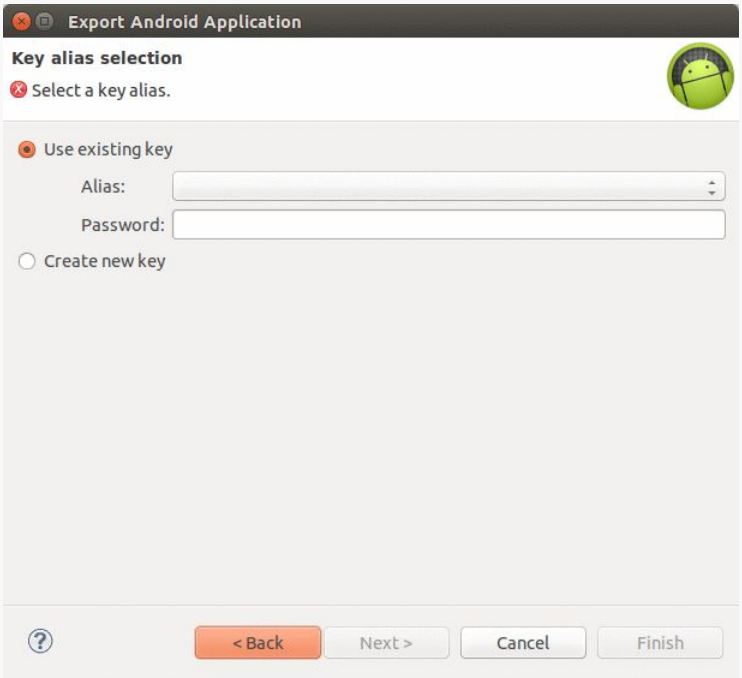


Figure 342: Eclipse Production App Wizard, Existing-Keystore Page

Once your keystore and key are set up or chosen, the last page of the wizard is where you indicate where the signed APK should be stored:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

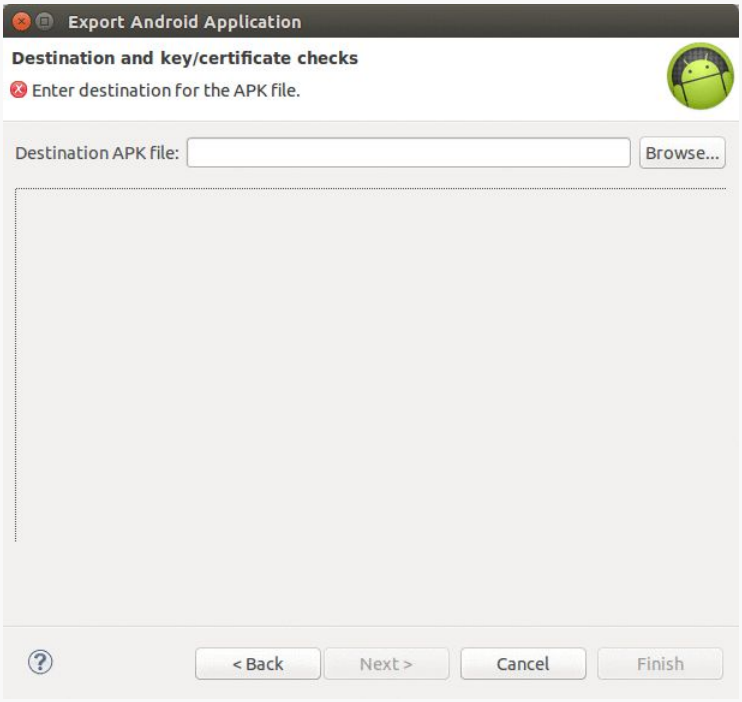


Figure 343: Eclipse Production App Wizard, Final Page

Providing a path and clicking Finish will export the signed production APK file.

Two Types of Key Security

There are two facets to securing your production key that you need to think about:

- You need to make sure nobody steals your production keystore and its password. If somebody does, they could publish replacement versions of your applications — since they are signed with the same key, Android will assume the replacements are legitimate.
- You need to make sure you do not lose your production keystore and its password. Otherwise, even *you* will be unable to publish replacement versions of your applications.

For solo developers, the latter scenario is more probable. There already have been **many** cases where developers had to rebuild their development machine and wound up with new keys, locking themselves out from updating their own applications. As with everything involving computers, having a solid backup regimen is highly recommended. In particular, consider a secure off-site backup, such as having your production keystore on a thumb drive in a bank safe deposit box.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For teams, the former scenario may be more likely. If more than one person needs to be able to sign the application, the production keystore will need to be shared, possibly even stored in the revision control system for the project. The more people who have access to the keystore, the more likely it is somebody will wind up doing something evil with it. This is particularly true for projects with public revision control systems, such as open source projects — developers might not think of the implications of putting the production keystore out for people to access.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Distribution

It is entirely possible that the user base for your app consists solely of yourself.

However, in most cases, you are going to be giving your app to others, free or for some sort of fee.

This chapter outlines things you will need to think about when distributing your app.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the chapter on signing your app.

Get Ready To Go To Market

While being able to sign your application reliably with a production key is necessary for publishing a production application, it is not sufficient. Particularly for the Play Store, there are other things you must do, or should do, as part of getting ready to release your application.

Versioning

You need to supply `versionCode` and `versionName` values in your `build.gradle` file (or in your `<manifest>` element in your `AndroidManifest.xml` file for Eclipse users). The value of `versionName` is what users and prospective users will see in terms of the label associated with your application version (e.g., "1.0.1", "System V", "Loquacious Llama"). More important, though, is the value of `versionCode`, which

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

needs to be an integer increasing with each release — that is how Android tells whether some edition of your APK is an upgrade over what the user currently has.

Application ID

You also need to make sure that your application ID is going to be unique. If somebody tries downloading your application onto their device, and some other application is already installed with that same package name, your application will fail to install.

Your application ID defaults to be the value of your `package` attribute in your `<manifest>` element in the manifest. You can override the application ID using `applicationId` properties in `defaultConfig` or a product flavor in `build.gradle`. You can also append an `applicationIdSuffix` tied to a build type in Gradle as well.

Since the manifest's `package` also provides the base Java package for your project, and since you hopefully named your Java packages with something based off of a domain name you own or something else demonstrably unique, this should not cause a huge problem.

Also, bear in mind that your application ID must be unique across all applications on the Play Store, should you choose to distribute that way.

Icon and Label

Your `<application>` element needs to specify `android:icon` and `android:name` attributes, to supply the name and icon that will be associated with the application in the My Applications list on the device and related screens. Your activities will inherit the icon if they do not specify icons of their own.

If you have graphic design skills, the Android developer site has guidelines for creating icons that will match other icons in the system.

Logging

In production, try to minimize unnecessary logging, particularly at low logging levels (e.g., debug). Remember that even if Android does not actually log the information, whatever processing is involved in making the `Log.d()` call will still be done, unless you arrange to skip the processing somehow. You could outright delete the extraneous logging calls, or wrap them in an `if()` test:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
if (BuildConfig.DEBUG) {
  Log.d(TAG, "This is what happened");
}
```

Here, `BuildConfig.DEBUG` is a `public static final boolean` value, supplied by Android, that indicates whether you are building for debug or production. Whether you adjust the definition by hand or by automating the build process is up to you. But, when `BuildConfig.DEBUG` is `false`, any work that would have been done to build up the actual `Log` invocation will be skipped, saving CPU cycles and battery life.

Conversely, error logs become even more important in production. Sometimes, you have difficulty reproducing bugs "in the lab" and only encounter them on customer devices. Being able to get stack traces from those devices could make a major difference in your ability to get the bug fixed rapidly.

First, in addition to your regular exception handlers, consider catching everything those handlers miss, notably runtime exceptions:

```
Thread.setDefaultUncaughtExceptionHandler(onBlooey);
```

This will route all uncaught exceptions to an `onBlooey` handler:

```
private Thread.UncaughtExceptionHandler onBlooey=
  new Thread.UncaughtExceptionHandler() {
  public void uncaughtException(Thread thread, Throwable ex) {
    Log.e(TAG, "Uncaught exception", ex);
  }
};
```

There, you can log it, raise a dialog if appropriate, etc.

Then, offer some means to get your logs off the device and to you, via email or a Web service. Some Android analytics firms, like Flurry, offer exception stack trace collection as part of their service. There are also open source projects that support this feature, such as ACRA.

Testing

As always, testing, particularly acceptance testing, is important.

Bear in mind that the act of creating the production signed version of your application could introduce errors, such as having the wrong Google Maps V2 API

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

key. Hence, it is important to do user-level testing of your application after you sign, not just before you sign, in case the act of signing messed things up. After all, what you are shipping to those users is the production signed edition — you do not want your users tripping over obvious flaws.

As you head towards production, also consider testing in as many distinct environments as possible, such as:

1. Trying more than one device, particularly if you can get devices with different display sizes
2. If you rely on the Internet, try your application with WiFi, with 3G, with EDGE/2G, and with the Internet unavailable
3. If you rely on GPS, try your application with GPS disabled, GPS enabled and working, and GPS enabled but not available (e.g., underground)

EULA

End-user license agreements — EULAs — are those long bits of legal prose you are supposed to read and accept before using an application, Web site, or other protected item. Whether EULAs are enforceable in your jurisdiction is between you and your qualified legal counsel to determine.

In fact, many developers, particularly of free or open source applications, specifically elect not to put a EULA in their applications, considering them annoying, pointless, or otherwise bad.

However, the Play Store developer distribution agreement has one particular clause that might steer you towards having a EULA:

> You agree that if you use the Store to distribute Products, you will protect the privacy and legal rights of users. If the users provide you with, or your Product accesses or uses, user names, passwords, or other login information or personal information, you must make the users aware that the information will be available to your Product, and you must provide legally adequate privacy notice and protection for those users. Further, your Product may only use that information for the limited purposes for which the user has given you permission to do so. If your Product stores personal or sensitive information provided by users, it must do so securely and only for as long as it is needed. But if the user has opted into a separate agreement with you that allows you or your Product to store or use personal or sensitive information directly related to your Product (not including

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

> other products or applications) then the terms of that separate agreement will govern your use of such information. If the user provides your Product with Google Account information, your Product may only use that information to access the user's Google Account when, and for the limited purposes for which, the user has given you permission to do so.

Hence, if you are concerned about being bound by what Google thinks appropriate privacy is, you may wish to consider a EULA just to replace their terms with your own.

Unfortunately, having a EULA on a mobile device is particularly annoying to users, because EULAs tend to be long and screens tend to be short.

Again, please seek professional legal assistance on issues regarding EULAs.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Advanced Gradle for Android Tips

There are *lots* of things you can do given a full scripting language as the basis for your build system. This chapter represents a collection of tips for things that you can do that go beyond stock capabilities provided by the Gradle for Android plugin.

NOTE: The projects demonstrated in this chapter are *not* set up to be used by Eclipse, as Eclipse does not support Gradle as of the time of this writing.

Prerequisites

Understanding this chapter requires that you have read the chapters that introduce Gradle and cover basic Gradle/Android integration, including both the legacy project structure and the new project structure. Having read the chapter on Gradle dependencies would also be a pretty good idea.

The chapter on using the NDK is a prerequisite for one section of this chapter, covering using Gradle with the NDK.

Gradle, DRY

Ideally, your build scripts do not repeat themselves any more than is logically necessary. For example, a project and sub-projects probably should use the same version of the build tools, yet by default, we define them in each `build.gradle` file. This section outlines some ways to consolidate this sort of configuration

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It's build.gradle All The Way Down

If you have sub-projects, you can have `build.gradle` files at each level of your project hierarchy. Your top-level `build.gradle` file is also applied to the sub-projects when they are built.

In particular, you can "pass data" from the top-level `build.gradle` file to sub-projects by configuring the `ext` object via a closure. In the top-level `build.gradle` file, you would put common values to be used:

```
ext {
   compileSdkVersion=19
}
```

(note the use of the = sign here)

Sub-projects can then reference `rootProject.ext` to retrieve those values:

```
android {
    compileSdkVersion rootProject.ext.compileSdkVersion
}
```

By this means, you can ensure that whatever needs to be synchronized at build time is synchronized, by defining it once.

Another way that a top-level `build.gradle` file can configure subprojects is via the `subprojects` closure. This contains bits of configuration that will be applied to each of the subprojects as a part of their builds.

The `HelloAIDL` sample project demonstrates this. The `build.gradle` in the overall project root (outside the `Client/` and `Service/` sub-projects) has a `subprojects` closure to define the code-signing rules for these two applications and common values for the two sub-projects:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}

subprojects {
    buildscript {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        repositories {
            mavenCentral()
        }
        dependencies {
            classpath 'com.android.tools.build:gradle:1.0.0'
        }
    }

    apply plugin: 'com.android.application'

    android {
        compileSdkVersion 19
        buildToolsVersion "21.1.2"

        signingConfigs {
            release {
                storeFile file('HelloAIDL.keystore')
                keyAlias 'HelloConfig'
                storePassword 'laser.yams.heady.testy'
                keyPassword 'fw.stabs.steady.wool'
            }
        }

        buildTypes {
            release {
                signingConfig signingConfigs.release
            }
        }
    }
}
```

The `subprojects` closure contains its own reference to the `android` plugin for Gradle, in addition to `android` closure for configuring the `signingConfigs` and `buildTypes`. Because this code is written in the root project's `build.gradle` file, `file()` references refer to the root project's directory, which is why `file('HelloAIDL.keystore')` will find the keystore in the root project's directory.

Note that `subprojects` applies to *all* sub-projects, which limits its utility. For example, a top-level project with one sub-project for an app and another sub-project for a library used by that app cannot readily use `subprojects`. That is because the library sub-project needs to configure the `com.android.library` plugin, while the application sub-project needs to configure the `com.android.application` plugin. The `subprojects` closure is only good for common configuration to apply to all sub-projects regardless of project type.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

gradle.properties

Another approach would be to add a `gradle.properties` file to your project root directory. Those properties are automatically read in and would be available up and down your project hierarchy.

Per-developer properties can go in a `gradle.properties` file in the user's Gradle home directory (e.g., `~/.gradle` on Linux), where they will not be accidentally checked into version control.

So, to achieve the synchronized `compileSdkVersion` value, you could have a `gradle.properties` file with:

```
COMPILE_SDK_VERSION=19
```

Then, your projects' `build.gradle` files could use:

```
android {
    compileSdkVersion COMPILE_SDK_VERSION
}
```

The Gradle/HelloProperties sample project illustrates this. It is a clone of the `HelloAIDL` sample application from earlier in this chapter, but one where we have a `gradle.properties` file in the root project's directory:

```
BUILD_TOOLS_VERSION=21.1.2
```

Here, we are defining a build tools version for use with the `buildToolsVersion` property in the `android` closure. The sub-projects use the `BUILD_TOOLS_VERSION` property that we defined in `gradle.properties` in their own `build.gradle` files, courtesy of a `subprojects` closure defined in the top-level `build.gradle` file:

```
        buildToolsVersion BUILD_TOOLS_VERSION
```

Custom Properties Files

You are also welcome to use your own custom properties files. For example, perhaps you want to use `gradle.properties` for properties that you are willing to put in version control (e.g., `BUILD_TOOLS_VERSION`), but you would *also* like to use a properties file to keep your code-signing details outside of your `build.gradle` file and out of version control.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Loading in custom properties files is slightly clunky, as it does not appear to be built into Gradle itself. However, you can take advantage of the fact that Gradle is backed by Groovy and use some ordinary Groovy code to load the properties.

This can also be seen in the `HelloProperties` sample project, where the `build.gradle` in the root project's directory uses a `signing.properties` file to isolate sensitive data:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}

subprojects {
    buildscript {
        repositories {
            mavenCentral()
        }
        dependencies {
            classpath 'com.android.tools.build:gradle:1.0.0'
        }
    }

    apply plugin: 'com.android.application'

    android {
        compileSdkVersion 19
        buildToolsVersion BUILD_TOOLS_VERSION

        def signingPropFile = rootProject.file('signing.properties')

        if (signingPropFile.canRead()) {
            def Properties signingProps = new Properties()

            signingProps.load(new FileInputStream(signingPropFile))

            signingConfigs {
                release {
                    storeFile rootProject.file('HelloAIDL.keystore')
                    keyAlias signingProps['KEY_ALIAS']
                    storePassword signingProps['STORE_PASSWORD']
                    keyPassword signingProps['KEY_PASSWORD']
                }
            }

            buildTypes {
                release {
                    signingConfig signingConfigs.release
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                }
            }
        }
    }
}
```

Let's look at the key lines, one at a time:

```
def signingPropFile = rootProject.file('signing.properties')
```

This statement grabs the `signing.properties` file from the root project and assigns it to the `signingPropFile` variable. Groovy, by default, is a dynamic language and does not use data types for its variables. Under the covers, `signingPropFile` is a `java.io.File` object, just like you are used to in ordinary Java/Android development.

```
if (signingPropFile.canRead()) {
}
```

Since `signingPropFile` is a `File`, we can call a `canRead()` method to confirm that the file exists and is readable.

```
def Properties signingProps = new Properties()
```

This creates an empty instance of a `java.util.Properties` object and assigns it to the `signingProps` variable.

```
signingProps.load(new FileInputStream(signingPropFile))
```

This creates a standard `java.io.FileInputStream` for the properties file, then passes it to the `load()` method on the `Properties` object, to read in the properties file.

```
keyAlias signingProps['KEY_ALIAS']
storePassword signingProps['STORE_PASSWORD']
keyPassword signingProps['KEY_PASSWORD']
```

These statements access properties from the `Properties` object, where Groovy has augmented `Properties` to support square-bracket syntax to access individual properties.

The author would like to thank Gabriele Mariotti for his blog post that, among other things, depicted this technique.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Environment Variables

Any environment variables with a prefix of `ORG_GRADLE_PROJECT_` will show up as global variables in your Gradle script. So, for example, you can access an environment variable named `ORG_GRADLE_PROJECT_foo` by accessing a `foo` variable in `build.gradle`.

If you would prefer to use environment variables without that prefix, you can call `System.getenv()`, passing in the name of the environment variable, to retrieve its value.

Note, however, that you may or may not have access to the environment variables that you think you should. Android Studio, for example, does not expose environment variables to Gradle for its builds, and so an environment variable that you can access perfectly well from the command line may not be available in the same `build.gradle` script when run from Android Studio.

Automating APK Version Information

Once Gradle for Android started catching on, one of the first things many developers raced to do was automate the `android:versionCode` and `android:versionName` properties from the manifest. Since those can be defined in a Gradle file (overriding values from any `AndroidManifest.xml` files), and since Gradle is backed by Groovy, it is possible to programmatically assign values to those properties.

This section outlines a few approaches to that problem.

Auto-Incrementing the versionCode

Since the `android:versionCode` is a monotonically increasing integer, one approach for automating it is to simply increment it on each build. While this may seem wasteful, two billion builds is a *lot* of builds, so a solo developer is unlikely to run out. Synchronizing such `versionCode` values across a team will get a bit more complex, but for an individual case (developer, build server, etc.), it is eminently doable using Groovy.

The `Gradle/HelloVersioning` sample project uses a `version.properties` file as the backing store for the version information:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (versionPropsFile.canRead()) {
        def Properties versionProps = new Properties()

        versionProps.load(new FileInputStream(versionPropsFile))

        def code = versionProps['VERSION_CODE'].toInteger() + 1

        versionProps['VERSION_CODE']=code.toString()
        versionProps.store(versionPropsFile.newWriter(), null)

        defaultConfig {
            versionCode code
            versionName "1.1"
            minSdkVersion 14
            targetSdkVersion 18
        }
    }
    else {
        throw new GradleException("Could not read version.properties!")
    }
```

First, we try to open a `version.properties` file and fail if it does not exist, requiring the developer to create a starter file manually:

```
VERSION_CODE=1
```

Of course, a more robust implementation of this script would handle this case and supply a starter value for the developer.

The script then uses the read-the-custom-properties logic illustrated in the preceding section to read the existing value... but it increments the old value by 1 to get the new code to use. The revised code is then written back to the properties file before it is applied in the `defaultConfig` closure.

In this case, the script throws a `GradleException` to halt the build if the `version.properties` file could not be found or otherwise could not be read.

Synchronizing the versionName... with the versionCode

If you do not want to automatically increment the `android:versionCode` value, you could use it to also create a matching `android:versionName` value. Jake Wharton illustrated this in a Google+ post, showing how you can build the `versionCode` up from parts representing the major, minor, and patch-level numbers, then use those same numbers to generate a standard dot-notation `versionName`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Synchronizing the versionName… with the APK File Name

You can also use the `android:versionCode` and `android:versionName` elsewhere in your Gradle build file, to apply to other aspects of your build. For example, Kevin Coppock posted a snippet of code showing how to embed your `versionName` into your compile APK's filename. The `HelloVersioning` sample uses a modified version of this same approach as part of its `buildTypes` closure:

```
    buildTypes {
        debug {
          applicationIdSuffix ".d"
          versionNameSuffix "-debug"
        }

        release {
            signingConfig signingConfigs.release
        }

        mezzanine.initWith(buildTypes.release)

        mezzanine {
            applicationIdSuffix ".mezz"
            debuggable true
            signingConfig signingConfigs.release
        }

        // from http://stackoverflow.com/a/27068573/115145

        applicationVariants.all { variant ->
            variant.outputs.each { output ->
                output.outputFile = new File(
                        output.outputFile.parent,
                        output.outputFile.name.replace(".apk",
"-${variant.versionName}.apk"))
            }
        }
    }
```

When defining the build types, we iterate over all application variants (build type/product flavor combinations), over each of the possible variant outputs (courtesy of splits), and modify the `outputFile` property of the variant to embed the `variant.versionName` value.

Adding to BuildConfig

The Android development tools have been code-generating the `BuildConfig` class for some time now. Historically, the sole element of that class was the `DEBUG` flag,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

which is `true` for a debug build and `false` otherwise. This is useful for doing runtime changes based upon build type, such as only configuring `StrictMode` in debug builds.

Nowadays, the Gradle for Android plugin also defines:

- `BUILD_TYPE`, which is the build type used to build this APK.
- `FLAVOR`, which is the product flavor used to build this APK.
- `PACKAGE_NAME`, which is the name that serves as the application ID (i.e., it includes build type suffixes and product flavor overrides). This is useful for cases where you cannot just call `getPackageName()` on a `Context` because you do not have a handy `Context`.
- `VERSION_CODE`, which is the version code derived from your manifest in conjunction with any overrides coming from your `build.gradle` file.
- `VERSION_NAME`, which is the version name derived from your manifest in conjunction with any overrides coming from your `build.gradle` file.

However, you can add your own data members to `BuildConfig`, by including a `buildConfigField` statement in the `defaultConfig` closure of your `android` closure:

```
android {
  defaultConfig {
    buildConfigField "int", "FOO", '5'
  }
}
```

You can use this to embed any sort of information you want into `BuildConfig`, so long as it is knowable at compile time.

Moreover, you can also have `buildConfigField` statements in build types. This would be useful if you have custom build types, beyond just debug and release, and you need runtime configuration for those. For example, you could put server URLs in `buildConfigField`, so your debug server is different from your integration test server, which in turn is different than your production server.

You can see this approach used in the Gradle/HelloBuildConfig sample project. Its `buildTypes` closure defines three different variations of a `SERVER_URL` field on the `BuildConfig` object:

```
    buildTypes {
        debug {
          applicationIdSuffix ".d"
          versionNameSuffix "-debug"
          buildConfigField "String", "SERVER_URL",
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
'"http://test.this-is-so-fake.com"'
        }

        release {
          signingConfig signingConfigs.release
          buildConfigField "String", "SERVER_URL",
'"http://prod.this-is-so-fake.com"'
        }

        mezzanine.initWith(buildTypes.release)

        mezzanine {
            applicationIdSuffix ".mezz"
            debuggable true
            buildConfigField "String", "SERVER_URL",
'"http://stage.this-is-so-fake.com"'
        }
    }
```

The Java code can refer to `BuildConfig.SERVER_URL` to retrieve this value. Since it is defined for all current build types, there will always be a value at compile time. Note, though, that if you add a build type, you need to ensure that it will have a `SERVER_URL` defined.

As of version 0.8 of the Gradle for Android plugin, if you redefine the same `buildConfigField` name, it replaces the previous value. So, in the `build.gradle` segment shown above, we define the `SERVER_URL` on the `release` build type *before* using `release` as the basis for the `mezzanine` build type. Right after the `mezzanine.initWith(buildTypes.release)` statement, the `mezzanine` build type has the same `buildConfigField` value for `SERVER_URL` as did `release`. But, we then replace that value in the `mezzanine` closure, to have a different server URL for `mezzanine` builds than we use for `release` or `debug` builds.

Down and Dirty with the DSL

What `build.gradle` does is build up an object model that describes a build process, in the form of defining tasks. Many times you can define the object model in a declarative fashion, with closures like `android` and `buildTypes` and `signingConfigs` and so on. However, as seen in this chapter, sometimes you need to get into Groovy scripting, and sometimes that scripting involves working with the Gradle for Android object model directly.

To help you understand what that object model looks like, the "Android Plug-in for Gradle" page contains a download button for "Plugin Command Reference". Despite

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the button's name, this is really documentation for the domain-specific language (DSL) published by the Gradle for Android plugin, including:

- all of the "configuration blocks", such as the `defaultConfig` closure
- all of the "DSL types", or the objects that are built up by those "configuration blocks", such as `BuildType` and `ProductFlavor`

At the present time, there is no link to an online hosted version of this documentation, just to the ZIP file to download the offline copy.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Testing

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

JUnit and Android

Presumably, you will want to test your code, beyond just playing around with it yourself by hand. Android offers several means of testing your app, covered in this next series of chapters.

The first Android SDK testing solution we will examine is the JUnit test framework. This is a standard Java unit testing framework. The Android SDK ships a version of JUnit, along with special test classes that will help you build test cases that exercise Android components, like activities and services.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

This chapter also assumes you have some familiarity with JUnit, though you certainly do not need to be an expert. You can learn more about JUnit at the JUnit site, from various books, and from the JUnit Yahoo forum.

Android Studio and Gradle for Android users may also wish to read the chapter on Gradle dependencies.

JUnit Basics

As a quick terminology refresher, for those who have not used JUnit much or have not used it recently:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- A **test case** is a Java class, inheriting (directly or indirectly) from `junit.framework.TestCase`, that represents a set of tests to run
- A **test method** is a method, in a test case, that begins with the keyword `test`, and contains code that tests something in some production code base
- A **test suite** is a collection of test cases

So, a test **suite** enumerates test **cases**, which in turn implement test **methods**.

A test case can also have `setUp()` and `tearDown()` methods, in addition to test methods. `setUp()` is called before each test method invocation, while `tearDown()` is called after each test method invocation. So, for a test case with `testOne()` and `testTwo()` methods, along with `setUp()` and `tearDown()` methods, the following series of method invocations will occur when that test case is run:

- `setUp()`
- `testOne()`
- `tearDown()`
- `setUp()`
- `testTwo()`
- `tearDown()`

Note that while the above list shows `testOne()` being called before `testTwo()`, developers should not rely upon the exact order in which test methods are invoked. Instead, the `setUp()` and `tearDown()` methods are specifically there to establish the environment for each test method run.

Pondering Gradle for Android

Android Studio uses Gradle for Android for building the APK. It *also* uses Gradle for Android for running JUnit tests.

Testing is one aspect of software development that is often automated. You could use a full-blown continuous integration server like Jenkins. Or, you could use something more casual like a Windows Scheduled Task to run tests daily in the early morning hours. Regardless, there is an above-average chance that you will want to run your tests both inside of and outside of Android Studio.

Hence, the instructions in this chapter will not only explain how to run tests from the IDE (both Android Studio and Eclipse), but also how to run tests from the command line using Gradle.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Where Your Test Code Lives

One common problem with testing is determining where the test code should reside, relative to the production code being tested. Ideally, these are not intermingled, as that would increase the odds that you might accidentally ship the testing code as part of your production app — at best, this increases your APK size; at worst, it could open up security flaws.

The approach that you take with JUnit and Android development depends on the environment in which your tests will run.

Android Studio and Gradle for Android

With Gradle-based projects, including those created for Android Studio, we have a dedicated *sourceset* for our instrumentation tests, named `androidTest`, where the code for those tests would reside.

NOTE: The name `androidTest` is for version 0.9.0 or higher of the Gradle for Android plugin. Older versions used `instrumentTest`.

As with any sourceset, `androidTest` can have Java code, resources, etc. It does not need an `AndroidManifest.xml` file, though, as that will be auto-generated.

The `Testing/JUnit-Gradle` sample project contains such an `androidTest` sourceset, and we will be looking at the Java code in there later in this chapter.

Your Android Studio projects should already start with an empty `androidTest` directory, into which you can add a `java/` subdirectory with your test code.

Your `build.gradle` file needs a couple of additional lines in its `defaultConfig` closure:

```
    defaultConfig {
        testApplicationId "com.commonsware.android.gradle.hello.test"
        testInstrumentationRunner "android.test.InstrumentationTestRunner"
    }
```

Here, we define:

- `testApplicationId` (formerly known as `testPackageName`), which is the name of the package to be used for the instrument test APK. For testing

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

apps, this needs to be a separate package name from the package name of the app. We will discuss testing Android library projects later in this chapter.

- `testInstrumentationRunner`, is the name of the Java class that implements the JUnit test runner to use for executing your tests. Usually, you will use `android.test.InstrumentationTestRunner`, though third-party libraries may offer alternative test runners.

Eclipse

Eclipse takes a different approach, where your test code resides in a completely separate Android test project.

An Android test project is a complete set of Android project artifacts: manifest, source directories, resources, etc. Much of its structure is identical to a regular project. In fact, the generated test project is all ready to go, other than not having any tests. For example, the `Testing/JUnit` project has a `tests/` subdirectory containing a test project.

To create a test project, from the standard Eclipse new-project dialog (File > New > Project), choose "Android Test Project".

The first page of the wizard will ask for your Eclipse settings, such as the project name:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

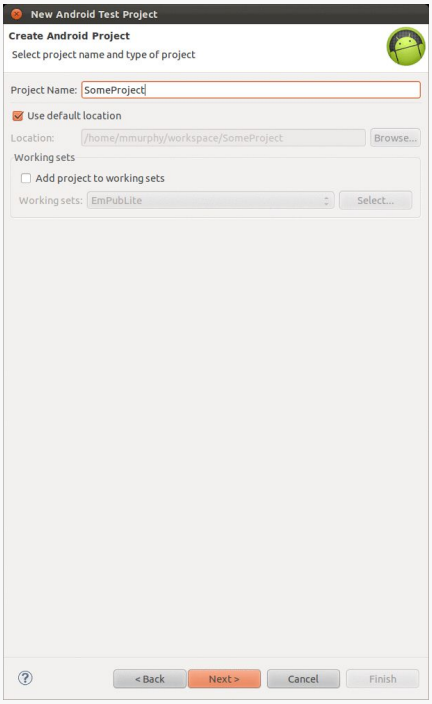


Figure 344: Eclipse Android Test Project Wizard, First Page

The second page of the wizard has you pick from one of your Android projects the one that you wish to test:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

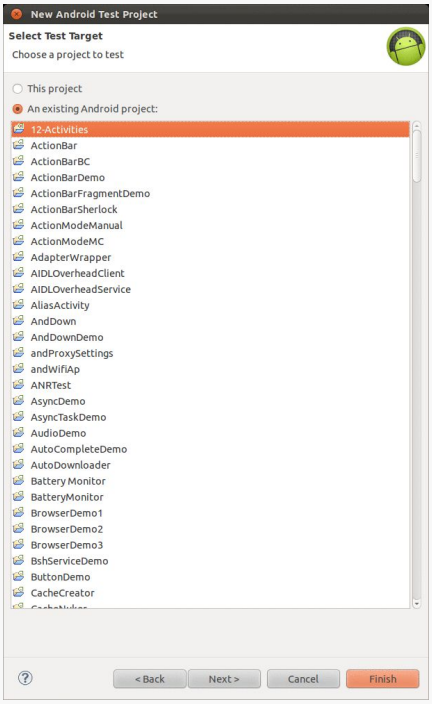


Figure 345: Eclipse Android Test Project Wizard, Second Page

The last page of the wizard lets you specify the build target, which will be based on the build target of the project you specified in the second page:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

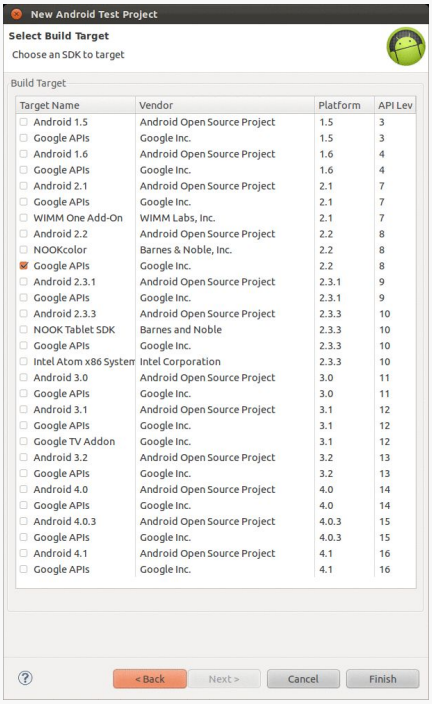


Figure 346: Eclipse Android Test Project Wizard, Third Page

Where Your Test Code Runs

Ordinarily, each code base (e.g., project) is packaged in its own APK and is executed in its own process.

In the case of so-called "instrumentation tests", like these JUnit tests, your test code and your production code are combined into a single process in a single copy of the virtual machine.

This will allow your JUnit test methods to access objects from your production code, such as your activities and their widgets.

However, this does limit instrumentation testing to be run from a developer's computer. You cannot package JUnit tests to be initiated from the device itself, except perhaps on rooted devices.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Writing Your Test Cases

A JUnit test project is made up of one (or potentially more) test suites, each comprising one (or usually more) test cases. A test case is a class, containing a series of test methods, designed to test some specific functionality. When a test case is run, JUnit:

- Creates an instance of the test case class
- Calls `setUp()`, where you can do any preperatory work
- Calls one of your test methods
- Calls `tearDown()` for post-test cleanup work
- Repeats the above steps for each test method

Hence, you need to write a series of test cases with test methods, and optionally `setUp()` and `tearDown()` as you see fit.

POJTCs (Plain Old JUnit Test Cases)

For tests that have nothing much to do with Android, you can use the standard JUnit `TestCase` base class. This works the same as JUnit would outside of Android, and is useful for testing business logic on POJOs (plain old Java objects) and the like.

For example, here is a test case that is, well, silly:

```
package com.commonsware.android.abf.test;

import junit.framework.TestCase;

public class SillyTest extends TestCase {
  protected void setUp() throws Exception {
    super.setUp();

    // do initialization here, run on every test method
  }

  protected void tearDown() throws Exception {
    // do termination here, run on every test method

    super.tearDown();
  }

  public void testNonsense() {
    assertEquals(1, 1);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

All we have is a single test method — `testNonsense()` that validates that 1 really does equal 1. Fortunately, this test usually succeeds. Our `TestCase` subclass (`SillyTest`) also implements `setUp()` and `tearDown()` for illustration purposes, as there is little preparation needed for our rigorous and demanding test method.

ActivityInstrumentationTestCase2

While ordinary JUnit tests are certainly helpful, they are still fairly limited, since much of your application logic may be tied up in activities, services, and the like.

To that end, Android has a series of `TestCase` subclasses that you can extend designed specifically to assist in testing these sorts of components.

The one most people focus on is `ActivityInstrumentationTestCase2`. As the name suggests, this class will run your activity for you, giving you access to the `Activity` object itself. You can then:

1. Access your widgets
2. Invoke public and package-private methods (more on this below)
3. Simulate key and touch events

Here are the steps to making use of `ActivityInstrumentationTestCase2`:

- Extend the class to create your own implementation. Since `ActivityInstrumentationTestCase2` is a generic, you need to supply the name of the activity being tested:

```
ActivityInstrumentationTestCase2<ActionBarFragmentActivity>
```

- In the constructor, when you chain to the superclass, supply the activity class itself.
- In `setUp()`, use `getActivity()` to get your hands on your `Activity` object, already typecast to the proper type (e.g., `ActionBarFragmentActivity`) courtesy of our generic. You can also at this time access any widgets, since the activity is up and running by this point.
- If needed, clean up stuff in `tearDown()`, no different than with any other JUnit test case.
- Implement test methods to exercise your activity.

For example, here is a short test case that exercises `ActionBarFragmentActivity`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.abf.test;

import android.test.ActivityInstrumentationTestCase2;
import android.test.TouchUtils;
import android.widget.ListView;
import com.commonsware.android.abf.ActionBarFragmentActivity;

public class DemoActivityTest extends
    ActivityInstrumentationTestCase2<ActionBarFragmentActivity> {
  private ListView list=null;

  public DemoActivityTest() {
    super(ActionBarFragmentActivity.class);
  }

  @Override
  protected void setUp() throws Exception {
    super.setUp();

    setActivityInitialTouchMode(false);

    ActionBarFragmentActivity activity=getActivity();

    list=(ListView)activity.findViewById(android.R.id.list);
  }

  public void testListCount() {
    assertEquals(25, list.getAdapter().getCount());
  }

  public void testKeyEvents() {
    sendKeys("4*DPAD_DOWN");
    assertEquals(4, list.getSelectedItemPosition());
  }

  public void testTouchEvents() {
    TouchUtils.scrollToBottom(this, getActivity(), list);
    getInstrumentation().waitForIdleSync();
    assertEquals(24, list.getLastVisiblePosition());
  }
}
```

In `setUp()`, we get access to the `ListView` that makes up the bulk of our UI, so we have access to that widget in any test method. We also have three test methods:

- `testListCount()`, where we check our `ListAdapter` in the `ListView` to make sure we have all 25 of our Latin words at the outset
- `testKeyEvents()`, where we simulate pressing the down arrow of a physical keyboard four times, then confirm that the selected row is the proper one

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `testTouchEvents()`, where we use `TouchUtils` and its `scrollToBottom()` method to scroll the `ListView` to the bottom, then confirm that we are actually at the bottom (the last visible row is the 25th row)

The `sendKeys()` method takes both individual key events via their `KeyEvent` constants (e.g., `KeyEvent.KEYCODE_BACK`) or a string containing space-delimited events, where the `KeyEvent.KEYCODE_` prefix is dropped for convenience. Also, a key event can be prefixed by a number and `*` to indicate that the key should be pressed that number of times. So, the `testKeyEvents()` call to `sendKeys("4*DPAD_DOWN")` will simulate four down arrow/D-pad presses in succession.

The `TouchUtils` class has a bunch of low-level methods for performing basic touch events — `scrollToBottom()` is about as sophisticated as they get. In the case of `scrollToBottom()`, the method takes the test case instance, the activity being tested, and the `ViewGroup` to be scrolled. Note that we call `getInstrumentation().waitForIdleSync()` after `scrollToBottom()`, to make sure that our test code blocks waiting for the work to be completed. This was not necessary for `sendKeys()`, as that is documented to have that behavior already.

If you are looking at your emulator or device while this test is running, you will actually see the activity launched on-screen. `ActivityInstrumentationTestCase2` creates a true running copy of the activity. This means you get access to everything you need; on the other hand, it does mean that the test case runs slowly, since the activity needs to be created and destroyed for each test method in the test case. If your activity does a lot on startup and/or shutdown, this may make running your tests a bit sluggish.

Note that our `ActivityInstrumentationTestCase2` resides in a different package than the `Activity` it is testing. This restricts us to pure black-box testing. If, however, we elected to put the test case in the same package as the activity, we could also call any package-private methods, for a test that is closer to white-box in style.

AndroidTestCase

For tests that only need access to your application resources, you can skip some of the overhead of `ActivityInstrumentationTestCase2` and use `AndroidTestCase`. In `AndroidTestCase`, you are given a `Context` and not much more, so anything you can reach from a `Context` is testable, but individual activities or services are not.

While this may seem somewhat useless, bear in mind that a lot of the static testing of your activities will come in the form of testing the layout: are the widgets

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

identified properly, are they positioned properly, does the focus work, etc. As it turns out, none of that actually needs an `Activity` object — so long as you can get the inflated `View` hierarchy, you can perform those sorts of tests.

Similarly, if you need to test business objects, but because they come from a database you need a `Context` for use with `SQLiteOpenHelper`, you could test those with an `AndroidTestCase`.

Here is a sample `AndroidTestCase`:

```
package com.commonsware.android.abf.test;

import android.test.AndroidTestCase;
import android.view.LayoutInflater;
import android.view.View;
import com.commonsware.android.abf.R;

public class DemoContextTest extends AndroidTestCase {
  private View field=null;
  private View root=null;

  @Override
  protected void setUp() throws Exception {
    super.setUp();

    LayoutInflater inflater=LayoutInflater.from(getContext());

    root=inflater.inflate(R.layout.add, null);
    root.measure(800, 480);
    root.layout(0, 0, 800, 480);

    field=root.findViewById(R.id.title);
  }

  public void testExists() {
    assertNotNull(field);
  }

  public void testPosition() {
    assertEquals(0, field.getTop());
    assertEquals(0, field.getLeft());
  }
}
```

Here, we manually inflate the contents of the `res/layout/add.xml` resource, and lay them out as if they were really in an activity, via calls to `measure()` and `layout()` to simulate a WVGA800 display. At that point, we can start testing the widgets inside of that layout, from simple assertions to confirm that they exist, to testing their size and position.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Other Test Cases

Android also offers various other test case base classes designed to assist in testing Android components, such as:

1. `ServiceTestCase`, used for testing services, as you might expect given the name
2. `ActivityUnitTestCase`, a `TestCase` that creates the `Activity` (like `ActivityInstrumentationTestCase`), but does not fully connect it to the environment, so you can supply a mock `Context`, a mock `Application`, and other mock objects to test out various scenarios
3. `ApplicationTestCase`, for testing custom `Application` subclasses

Your Test Suite

You will want to organize your test cases into one or more test suites. Many test projects have a single suite. However, elaborate test projects may have different suites for different situations, each representing some subset of the total roster of test cases defined in the project.

The simplest way to set up a test suite is to use Android's built-in `TestSuiteBuilder` class, that pulls in a series of test cases based upon package name, such as the `FullSuite` class in our sample test project:

```
package com.commonsware.android.abf.test;

import android.test.suitebuilder.TestSuiteBuilder;
import junit.framework.Test;
import junit.framework.TestSuite;

public class FullSuite extends TestSuite {
  public static Test suite() {
    return(new TestSuiteBuilder(FullSuite.class)
                .includeAllPackagesUnderHere()
                .build());
  }
}
```

Here, we are telling Android to find everything in this package (and sub-packages, if there were any) that implements `TestCase` and include it in the suite. Hence, organizing multiple suites would be a matter of organizing their test cases into separate packages and creating `TestSuite` classes per package.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Running Your Tests

Writing tests is nice. Running tests is nicer. Hence, it would be useful if we could run our JUnit tests. The exact procedure for doing this varies by environment.

Android Studio

At the moment, running tests from Android Studio is a bit of a pain.

First, you must define a new run configuration. To do that, choose Run > "Edit Configurations" from the main menu. That will bring up a dialog showing your current run configurations:

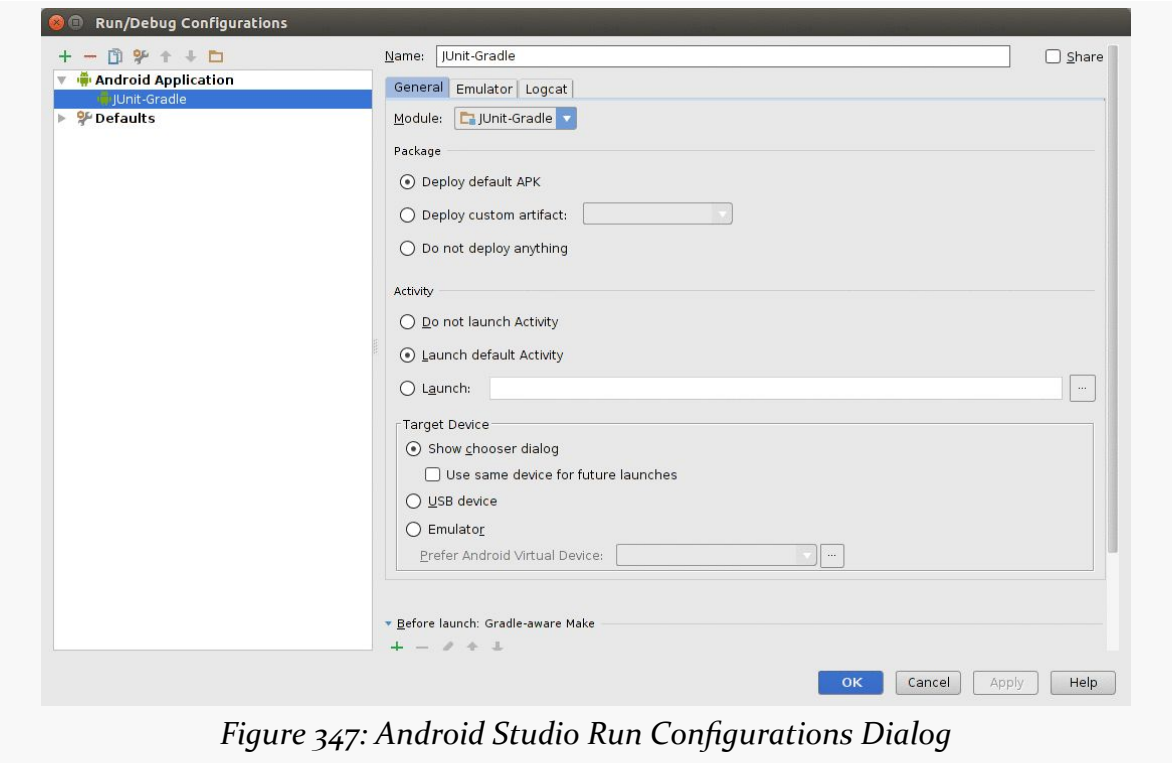


Figure 347: Android Studio Run Configurations Dialog

Towards the upper-left corner of the dialog, you will see a green plus sign. Tapping that will drop down a list of configuration types to choose from:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 348: Android Studio, Adding a New Run Configuration

Choose "Android Tests", and a new empty configuration will be set up for you. You can name it whatever you want via the "Name" field (e.g., "Unit Tests"). Choose your project's module that you wish to test in the "Module" drop-down (e.g., app). You can also choose the scope of the testing (e.g., "All in Module"), where to run the tests (e.g., "Show chooser dialog"), plus other settings.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

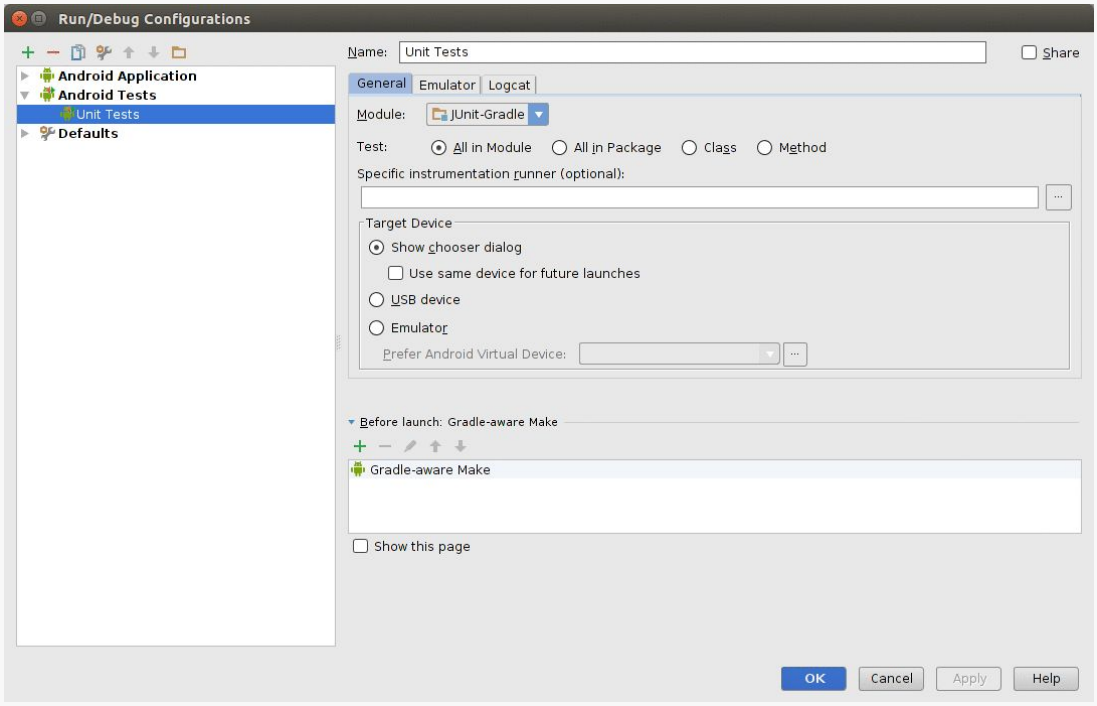


Figure 349: Android Studio, Showing New "Unit Tests" Run Configuration

At that point, you can choose your run configuration from the drop-down to the left of the "play" button in the toolbar:

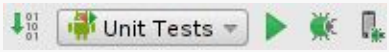


Figure 350: Android Studio Toolbar, Showing "Unit Tests" Run Configuration

Running that configuration will run your tests. The output will be shown in the Run view, normally docked in the bottom of your IDE window:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

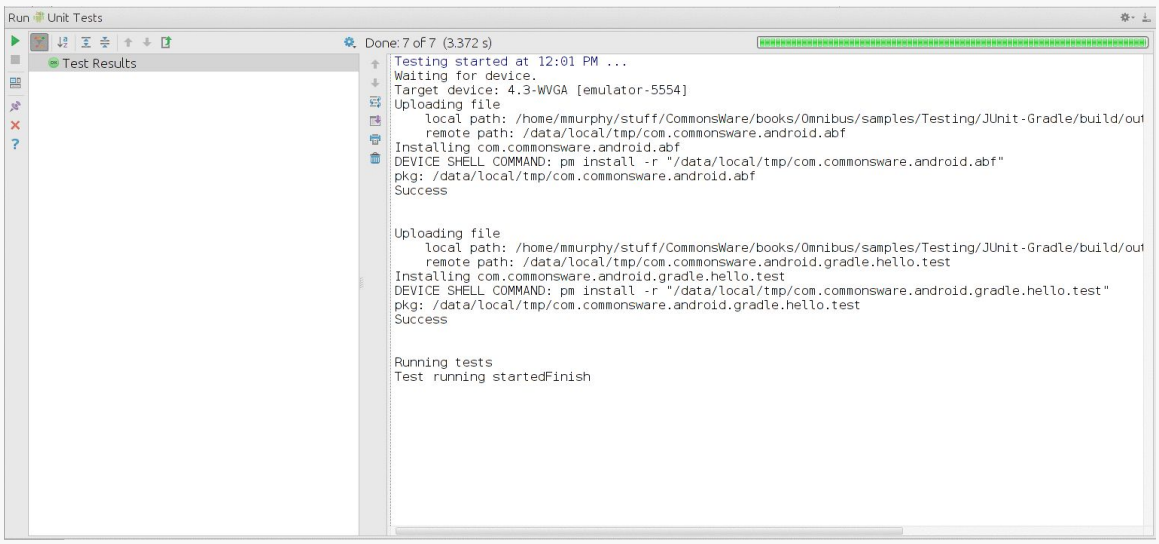


Figure 351: Android Studio, Showing Run Unit Tests Results

If a test fails an assertion or crashes, the test results will show the test case and test method that failed, along with the associated stack trace:

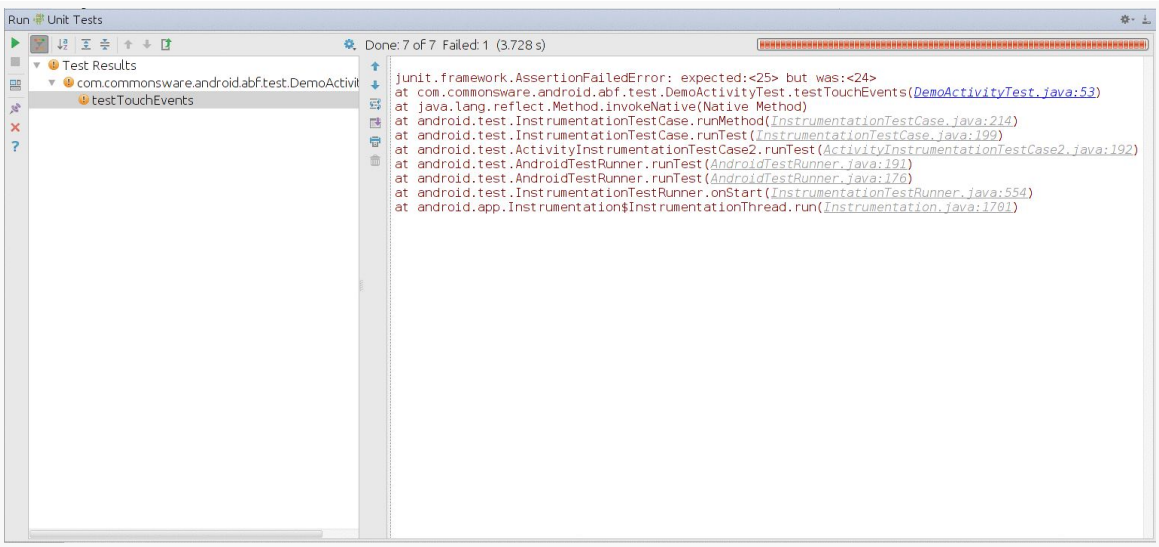


Figure 352: Android Studio, Showing Run Unit Tests Results With a Failure

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Gradle for Android

The primary Gradle task that you will use related to testing is `connectedCheck`. This task will build the main app, then, build the test app (using a generated manifest to go along with the code from your `androidTest` sourceset).

At that point, the task will iterate over *all* compatible connected devices and running emulator instances. For each such Android environment, the task will install both apps, run the tests, and uninstall both apps.

Raw test results, in XML format, will be written to `build/outputs/androidTest-results/connected`. These will primarily be of interest to toolsmiths, such as those adding support for Android Gradle-based builds to continuous integration (CI) servers.

For others, the HTML reports will be of greater use. These will be written to `build/outputs/reports/androidTests/connected`, with an `index.html` file serving as your entry point. These will show the results of all of your tests, with hyperlinked pages to be able to "drill down" into the details, such as to investigate failed tests.

Eclipse

You run a test project in Android the same way that you run a regular project. However, when you get the "Run As" dialog, choose "Android JUnit Test". By default, it will run all of your tests.

However, if you have a single `TestCase` class selected in your Package Explorer, Android can run just that single test case, rather than the full thing — again, choose "Android JUnit Test" in the "Run As" dialog:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

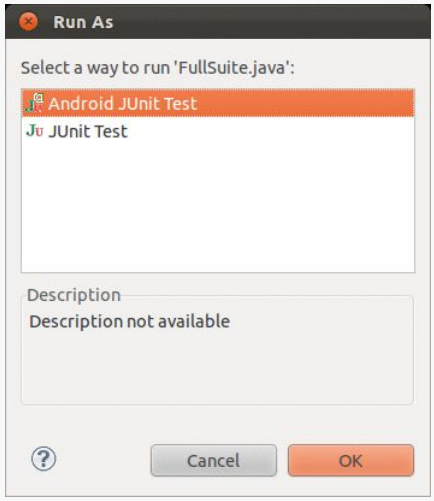


Figure 353: Eclipse "Run As" Dialog for JUnit Test

The results will be displayed in a JUnit view added to your Eclipse workspace, showing the successful and failed tests:

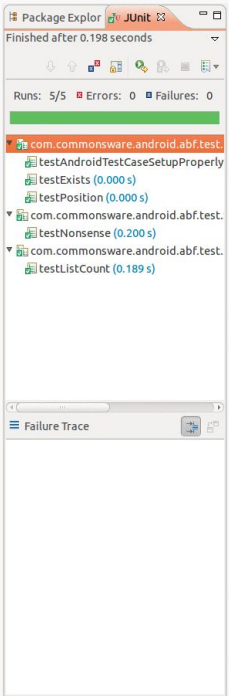


Figure 354: Eclipse Android JUnit Test Results

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Testing Android Library Projects

The above procedures are aimed at testing Android application projects. If you are creating an Android library project, you can also use JUnit for testing. However, there will be slight variations in the procedure, once again depending upon environment.

Android Studio and Gradle for Android

A Gradle-built Android library project can have an `androidTest` sourceset, just like a regular app. And, a Gradle-built Android library project can be tested via the `connectedCheck` task. However, that task will create and install a single APK, consisting of the code from the `androidTest` sourceset combined with the library project's own code.

From the standpoint of what you do as a developer, though, it works just like testing an app: add your test cases to the `androidTest` source set and use `connectedCheck` to run the tests.

Eclipse

In Eclipse, to test an Android library project, you create a separate test project and set it up to "this project" (i.e., the package being tested was the test project's own package). This is an option shown on the second page of the new test project wizard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

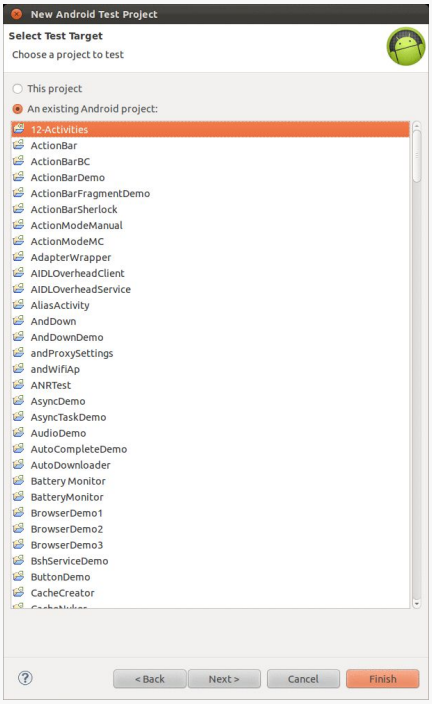


Figure 355: Eclipse Android Test Project Wizard, Second Page

Toggle the radio button at the top to "This project" to set up a test project that tests itself. You then add the Android library project to the test project and write test cases to exercise the library.

Test Dependencies

Sometimes, your test code will have no dependencies, other than the app (or library) being tested.

Sometimes, your test code will have its own dependencies, such as on libraries like Robotium. You need those dependencies for your test code, but you do not want those dependencies to be part of the production app.

To handle this, Gradle for Android (and Android Studio by extension) supports an `androidTestCompile` statement in the `dependencies` closure. Where `compile` states dependencies for the entire project; `androidTestCompile` states dependencies only for the instrumentation testing:

```
dependencies {
    // regular dependencies
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    compile 'com.android.support:support-v4:19.0.0'
    // ...

    // test dependencies
    androidTestCompile 'com.jayway.android.robotium:robotium-solo:4.3.+'
    // ...
}
```

This works similarly to specifying dependencies for a particular build type.

The equivalent in Eclipse would be to add the dependencies to the test project, such as copying a JAR into the test project's `libs/` directory.

Testing Legacy Project Structures with Gradle for Android

Suppose that you have a project that you still need to be buildable using Eclipse, yet you want to start using Gradle. For ordinary projects, using an Eclipse-exported `build.gradle` file will have a setup that supports this. However, your test project will be a standalone Android project, not merely some sourceset, as Eclipse does not know what a "sourceset" even is.

However, just as you can teach the `main` sourceset where it can find its files, you can do the same thing with the `androidTest` sourceset.

For example, suppose that you have an Android project, one that has a test project located in `tests/`. Both projects follow the legacy project structure. The `android` closure of your `build.gradle` file could look like this:

```
android {
    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            aidl.srcDirs = ['src']
            renderscript.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }

        androidTest {
            java.srcDirs = ['tests/src']
            resources.srcDirs = ['tests/src']
            aidl.srcDirs = ['tests/src']
            renderscript.srcDirs = ['tests/src']
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
            res.srcDirs = ['tests/res']
            assets.srcDirs = ['tests/assets']
        }
    }
}
```

(from Xavier Ducrohet's answer on Stack Overflow)

Here, we teach both the `main` and the `androidTest` sourcesets where their respective files are. In the case of the `androidTest` sourceset, we point it to files in the `tests/` subdirectory.

Given this, Gradle can run the tests, just as IDEs can.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Testing with JUnit4

Since the beginning of Android app development, we had unit testing capability via the `AndroidTestRunner`. However, this was based on JUnit3, which nowadays is a bit old.

In late 2014, the `AndroidJUnitRunner` (supporting JUnit4) was added to the Android Support package. This wa initially developed outside of the Android tools team, by a separate group in Google devoted to test automation.

In this chapter, we will review how to apply the `AndroidJUnitRunner` to run JUnit4 tests for our Android apps.

Prerequisites

Understanding this chapter requires that you have read the chapter on JUnit and the chapter on Gradle dependencies.

The AndroidJUnitRunner

JUnit uses test runners to execute our tests. Historically, most developers used `AndroidTestRunner`, as that was what was in the Android SDK and what was officially supported. However, in principle, JUnit test runners do not need to be part of the firmware. `AndroidJUnitRunner` is a great example of this, as it comes as part of the Android Support package, not as part of Android itself.

The primary reason to use `AndroidJUnitRunner` is to be able to use JUnit4 and Espresso tests. That being said, and subject to a few caveats, it is entirely possible for `AndroidJUnitRunner` to run existing JUnit3 tests. This is very useful, given that

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

rewriting an entire test suite to move *en masse* to JUnit4 may be very time-consuming.

The Testing/AndroidJUnitRunner sample project is a modified fork of the Testing/JUnit-Gradle sample project seen in the chapter on JUnit, with the minimum changes needed to get the tests to run successfully with AndroidJUnitRunner.

Note that this sample app only supports Android Studio. There does not appear to be an official recipe for using AndroidJUnitRunner on Eclipse.

Gradle Changes

To use AndroidJUnitRunner, you will need to make a few adjustments to your module's build.gradle file.

First, you need to add a test dependency — a dependency that will only be used as part of instrumentation testing. That can be accomplished via an androidTestCompile statement in the dependencies closure, instead of a compile statement, to limit the scope of the dependency to case where the androidTest sourceset is in use.

Specifically, AndroidJUnitRunner resides in the com.android.support.test:testing-support-lib artifact in the Android Support Repository, which by now you probably have installed via the SDK Manager.

```
dependencies {
    androidTestCompile 'com.android.support.test:testing-support-lib:0.1'
}
```

As of the time of this writing, the latest version of this artifact is 0.1.

However, this artifact also depends upon some other artifacts available in public repositories, like Maven Central or Bintray's JCenter. If you have one of those set up already — and a typical Android Studio project will via the top-level build.gradle file — then Gradle will be able to search those repositories for the dependencies. If you do not already have such a repository set up, you will need to do so via the repositories closure, such as is the case in the sample app:

```
repositories {
    mavenCentral() // required for testing-support-lib dependencies
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

And, because multiple dependencies bundle a `LICENSE.txt` file, and even though we have no use for the `LICENSE.txt` file in the build, the Gradle for Android plugin still complains about duplicate editions of this file. This requires us to filter this out, via a `packagingOptions` closure inside the `android` closure, to `exclude` the offending file:

```
android {
    compileSdkVersion 19
    buildToolsVersion "21.1.2"

    defaultConfig {
        testApplicationId "com.commonsware.android.gradle.hello.test"
        testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
    }

    packagingOptions {
        exclude 'LICENSE.txt' // required for no good reason...
    }
}
```

Switching Test Runners

Then, every place where we specify a test runner, we can use `AndroidJUnitRunner` instead of `AndroidTestRunner`. This includes the `testInstrumentationRunner` defined in the `defaultConfig` closure, shown above.

When running tests within Android Studio, the run configuration can specify the instrumentation runner to use:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

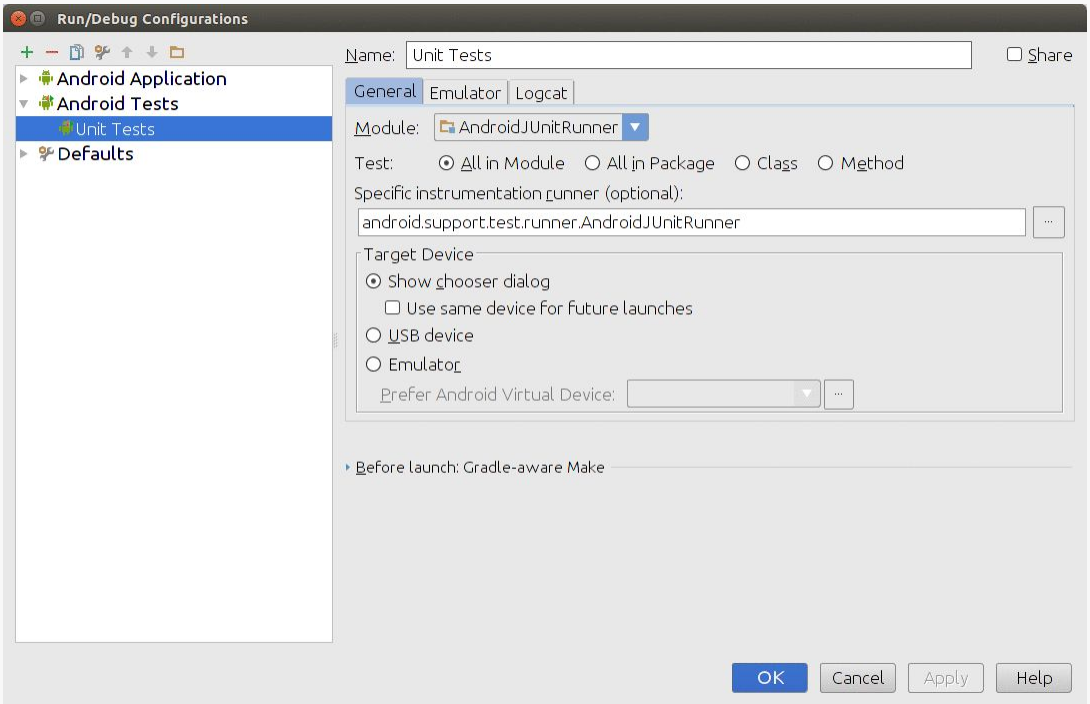


Figure 356: Android Studio Run Configuration Dialog, Showing AndroidJUnitRunner

Removing Your Suites

The `JUnit-Gradle` sample app from the chapter on JUnit had a `FullSuite` implementation of JUnit3's `TestSuite`. This does not appear to be supported by `AndroidJUnitRunner` at present.

Results

The results of running the `AndroidJUnitRunner` tests — other than the aforementioned suite limitation — should be identical to what you would have with using the classic `AndroidTestRunner`.

JUnit4

JUnit4, introduced in 2006, is a substantial revision to JUnit3, offering greater flexibility in how you set up your test code. JUnit4 is the current basis for JUnit, and so with `AndroidJUnitRunner`, Android developers are more up to date with respect to which version of JUnit they can use.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This book does not attempt to cover all aspects of JUnit4. For that, you are encouraged to read the JUnit documentation or other books on Java testing. This chapter will cover some of the basics of using JUnit4 tests, plus some of the issues with using JUnit4 tests in Android.

The Basics of 4

JUnit4 has the same spirit of testing as does JUnit3, but the actual mechanics are significantly different.

The Class

In JUnit3, we had to inherit from a magic `TestCase` class, supplied by JUnit.

In JUnit4, that is no longer the case (pun lightly intended).

Any Java class can serve as a test case, so long as it has a zero-argument public constructor and is known to the test runner that it contains tests to be run. In the case of `AndroidJUnitRunner`, that comes via a `@RunWith(AndroidJUnit4.class)` annotation on the class, to signal that this class contains tests:

```
@RunWith(AndroidJUnit4.class)
public class ICanHazTests {
  // test code goes here
}
```

Identifying Test Methods

In JUnit3, test methods were literally methods whose names began with `test`. That, coupled with the reliance upon `TestCase` as a base class, allowed JUnit3 to use reflection to find the subclasses of `TestCase`, then find all of the test methods within those test cases.

In JUnit4, a test method is any public method that is annotated with the `@Test` annotation:

```
@RunWith(AndroidJUnit4.class)
public class ICanHazTests {
  @Test
  public void kThxBye() {
    // do some testing
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

While there is nothing stopping you from prefixing the method name with `test`, it is superfluous and usually skipped.

Asserting Results

Since we no longer have access to inherited assertion methods from `TestCase`, JUnit4 supplies an `Assert` class with static assertion methods that we can employ:

```
@RunWith(AndroidJUnit4.class)
public class SillyTest {
  @Test
  public void thisIsReallySilly() {
    Assert.assertEquals("bit got flipped by cosmic rays", 1, 1);
  }
}
```

`assertEquals()` takes either two parameters of the same type for comparison (e.g., two `int` values), or three parameters, where the first is a custom assertion failure message.

There are countless other methods on `Assert` (e.g., `assertNotNull()`) for testing objects, collections, etc.

Setup and Teardown

In JUnit3, we could have methods with the magic names of `setUp()` and `tearDown()`. These would be called before (`setUp()`) and after (`tearDown()`) each test method invocation, to set up test "fixtures" (i.e., starter data to be available to all test methods).

In JUnit4, you can name the methods whatever you want, but you need to annotate them with `@Before` (replacing `setUp()`) and `@After` (replacing `tearDown()`). JUnit4 also adds static `@BeforeClass` and `@AfterClass` methods, which are invoked once for the entire test case, designed for setting up immutable starter data for test methods and avoiding the overhead of doing that work on each test method invocation.

The Testing/JUnit4 sample project is a reworking of the `Testing/AndroidJUnitRunner` sample project, this time with all of the test cases set up for JUnit4. Here, `SillyTest` has more flexibility — showing off the implementation of `@BeforeClass` and `@AfterClass`, for example — but is still equally silly:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.abf.test;

import android.support.test.runner.AndroidJUnit4;
import junit.framework.Assert;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(AndroidJUnit4.class)
public class SillyTest {
  @BeforeClass
  static public void doThisFirstOnlyOnce() {
    // do initialization here, run once for all SillyTest tests
  }

  @Before
  public void doThisFirst() {
    // do initialization here, run on every test method
  }

  @After
  public void doThisLast() {
    // do termination here, run on every test method
  }

  @AfterClass
  static public void doThisLastOnlyOnce() {
    // do termination here, run once for all SillyTest tests
  }

  @Test
  public void thisIsReallySilly() {
    Assert.assertEquals("bit got flipped by cosmic rays", 1, 1);
  }
}
```

Requirements and Backwards Compatibility

One major limitation with `AndroidJUnitRunner`, with respect to JUnit4, is that while your overall suite of test code can be a mix of JUnit3 and JUnit4, you cannot have a single test *class* mix the approaches. Either the class should extend `TestCase` and follow JUnit3 rules (e.g., test methods prefixed by `test`) *or* use the JUnit4 annotations, but not both.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Reworking Android-Specific Test Cases

Standard JUnit4 stuff works fine for testing POJOs or other functionality that is not tied to Android constructs like a `Context`.

The `testing-support-lib` artifact does not contain equivalents for classes like `ActivityInstrumentationTestCase2` (or other classes that inherit from `InstrumentationTestCase`) or `AndroidTestCase`. If you wish, you are welcome to stick with JUnit3 syntax for those. If, however, you would prefer to use JUnit4 syntax, you can certainly try.

ActivityInstrumentationTestCase2

The documentation for `AndroidJUnitRunner` contains instructions — albeit incomplete ones — for using `ActivityInstrumentationTestCase2` with JUnit4 syntax. The same basic approach may work for other subclasses of `InstrumentationTestCase`. This is illustrated in the revised version of `DemoActivityTest` in the sample app.

First, as with any JUnit4 test class, you have to add the `@RunWith(AndroidJUnit4.class)` annotation to the class itself:

```
@RunWith(AndroidJUnit4.class)
public class DemoActivityTest extends
    ActivityInstrumentationTestCase2<ActionBarFragmentActivity> {
```

If you did not have a `setUp()` method, you need to add one. If you did have a `setUp()` method, you will need to change it. Specifically:

- It now needs to be `public`
- It now needs the `@Before` annotation, as JUnit4 does not pay any attention to the `setUp()` method name
- It needs to have the `injectInstrumentation(InstrumentationRegistry.getInstrumentation())` statement added, shortly after `super.setUp()`, to hook in the instrumentation offered via `AndroidJUnitRunner`

```
  @Before
  @Override
  public void setUp() throws Exception {
    super.setUp();

    injectInstrumentation(InstrumentationRegistry.getInstrumentation());
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    setActivityInitialTouchMode(false);

    ActionBarFragmentActivity activity=getActivity();

    list=(ListView)activity.findViewById(android.R.id.list);
  }
```

If you did not have a `tearDown()` method, you need to add one. If you did have a `tearDown()` method, you will need to change it, with the same basic steps as the first two we did for `setUp()`:

- It now needs to be `public`
- It now needs the `@After` annotation, as JUnit4 does not pay any attention to the `tearDown()` method name

The required `setUp()` and `tearDown()` is because we can no longer rely upon JUnit4 magically calling those methods, and `ActivityInstrumentationTestCase2` expects them to be called.

You also need to add the `@Test` annotation to any test method, as with any other JUnit4 test class. Plus, you will need to switch to the `Assert` class' assertions, rather than any inherited ones.

So, the revised `DemoActivityTest` looks like:

```
package com.commonsware.android.abf.test;

import android.support.test.InstrumentationRegistry;
import android.support.test.runner.AndroidJUnit4;
import android.test.ActivityInstrumentationTestCase2;
import android.test.TouchUtils;
import android.widget.ListView;
import com.commonsware.android.abf.ActionBarFragmentActivity;
import junit.framework.Assert;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(AndroidJUnit4.class)
public class DemoActivityTest extends
    ActivityInstrumentationTestCase2<ActionBarFragmentActivity> {
  private ListView list=null;

  public DemoActivityTest() {
    super(ActionBarFragmentActivity.class);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Before
  @Override
  public void setUp() throws Exception {
    super.setUp();

    injectInstrumentation(InstrumentationRegistry.getInstrumentation());
    setActivityInitialTouchMode(false);

    ActionBarFragmentActivity activity=getActivity();

    list=(ListView)activity.findViewById(android.R.id.list);
  }

  @After
  public void tearDown() throws Exception {
    super.tearDown();
  }

  @Test
  public void listCount() {
    Assert.assertEquals(25, list.getAdapter().getCount());
  }

  @Test
  public void keyEvents() {
    sendKeys("4*DPAD_DOWN");
    Assert.assertEquals(4, list.getSelectedItemPosition());
  }

  @Test
  public void touchEvents() {
    TouchUtils.scrollToBottom(this, getActivity(), list);
    getInstrumentation().waitForIdleSync();
    Assert.assertEquals(24, list.getLastVisiblePosition());
  }
}
```

Note that you will get warnings on each of your `@Test` methods, complaining about how you are using that annotation on a class that inherits from JUnit3's `TestCase`. This is intentional behavior on our part. However, there is no obvious way to suppress this warning.

The Wharton Alternative

However, this is not your only option. In fact, Jake Wharton appears to not like this option very much, and he has offered an alternative solution: `ActivityRule`.

In JUnit4, a `Rule` is a way of encapsulating setup and teardown work in a way that does not force the use of inheritance from base classes. Mr. Wharton's

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`ActivityRule` encapsulates a bit of the logic from `ActivityInstrumentationTestCase2` for setting up and finishing an activity for every test method.

This does simplify testing for some cases:

```
@RunWith(AndroidJUnit4.class)
public final class TrivialTest {
  @Rule public final ActivityRule<YourActivityGoesHere> main
    =new ActivityRule(YourActivityGoesHere.class);

  // your test methods go here
}
```

The test methods can refer to `main` to call the `get()` method, which returns the `Activity` instance being tested (already cast to be your activity class). This replaces the `getActivity()` call you used to inherit from `ActivityInstrumentationTestCase2`.

However, it would take more code to cover everything that is lost from `ActivityInstrumentationTestCase2`. There are other methods, like `sendKeys()`, that we got by inheriting from `ActivityInstrumentationTestCase2` that we lack. Also, `TouchUtils` expects an `InstrumentationTestCase`, and that would need to be rewritten to work without one.

The sample app has a copy of `ActivityRule`, plus a `DemoActivityRuleTest` demonstrating its use. The latter has a copy of `sendKeys()` from the Android Open Source Project to allow that particular test method to continue to work.

AndroidTestCase

`AndroidTestCase` and its subclasses are simpler to retrofit. These can be converted into ordinary JUnit4 test classes. Anywhere you used to call `getContext()` in an `AndroidTestCase` can be replaced with a call to `InstrumentationRegistry.getTargetContext()`, which will give you a `Context` in the app being tested.

The `DemoContextTest`, therefore, turns into:

```
package com.commonsware.android.abf.test;

import android.support.test.InstrumentationRegistry;
import android.support.test.runner.AndroidJUnit4;
import android.test.AndroidTestCase;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.view.LayoutInflater;
import android.view.View;
import com.commonsware.android.abf.R;
import junit.framework.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(AndroidJUnit4.class)
public class DemoContextTest {
  private View field=null;
  private View root=null;

  @Before
  public void init() {
    LayoutInflater inflater=LayoutInflater
        .from(InstrumentationRegistry.getTargetContext());

    root=inflater.inflate(R.layout.add, null);
    root.measure(800, 480);
    root.layout(0, 0, 800, 480);

    field=root.findViewById(R.id.title);
  }

  @Test
  public void exists() {
    Assert.assertNotNull(field);
  }

  @Test
  public void position() {
    Assert.assertEquals(0, field.getTop());
    Assert.assertEquals(0, field.getLeft());
  }
}
```

The `InstrumentationRegistry` also has `getInstrumentation()` (which returns the `Instrumentation` object that we are using for testing) and `getContext()` (which returns the `Context` for our test code's package).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

MonkeyRunner and the Test Monkey

Many GUI environments have some means or another of "fuzz" or "bash" testing, where some test driver executes a bunch of random input, in hopes of catching errors (e.g., missing validation logic). Android offers the Test Monkey for this.

Many GUI environments have some means or another of scripting GUI events from outside the application itself, to simulate button clicks or touch events. Android offers MonkeyRunner for this.

As the names suggest, there is a bit of commonality in their implementation. And, as you might expect, there is a bit of commonality in their coverage in this book — we will examine both MonkeyRunner and the Test Monkey in this chapter.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

MonkeyRunner

MonkeyRunner is a means of creating test suites for Android applications based off of scripted UI input. Rather than write a series of JUnit test cases or the like, you create Jython (JVM implementation of Python) scripts that run commands to install apps, execute GUI events, and take screenshots of results.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Writing a MonkeyRunner Script

The primary object you will work with in a MonkeyRunner script is a `MonkeyDevice`, which represents your connection to the device or emulator that you are testing. You obtain a `MonkeyDevice` by calling `waitForConnection()` on `MonkeyRunner`; this will return once it has established a connection.

From there, `MonkeyDevice` lets you send events to the device or emulator:

- `installPackage()` allows you to install an APK from your development machine, and `removePackage()` allows you to get rid of it
- `startActivity()` and `broadcastIntent()` allow you to start up components of your app
- `press()` to simulate key events, including QWERTY keys, standard device keys like BACK, D-pad/trackball events, and anything else represented by a standard Android `KeyEvent`
- `type()` to simulate entering a whole string, as a simplification over calling `press()` once per letter
- `touch()` and `drag()` let you simulate touch events
- and so on

The biggest limitation is in getting data out of the device, to determine if your test worked successfully. Your options are:

- `takeSnapshot()`, which will capture a screenshot that you can save to disk, compare with other screenshots, etc.
- `shell()` executes `adb shell` commands, returning any results
- ...and that's about it

Unlike JUnit-based testing, you have no visibility into the activity beyond what appears on the screen — you cannot inspect widgets, call methods, or the like.

For example, here is a script that installs an app, runs an activity from it, and presses the down button on the D-pad three times:

```
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice

device = MonkeyRunner.waitForConnection()
device.installPackage('bin/JUnitDemo.apk')
device.startActivity(component='com.commonsware.android.abf/
com.commonsware.android.abf.ActionBarFragmentActivity')
device.press('KEYCODE_DPAD_DOWN', MonkeyDevice.DOWN_AND_UP)
device.press('KEYCODE_DPAD_DOWN', MonkeyDevice.DOWN_AND_UP)
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
device.press('KEYCODE_DPAD_DOWN', MonkeyDevice.DOWN_AND_UP)
# result = device.takeSnapshot()
# result.writeToFile('tests/monkey_sample_shots/test1.png', 'png')
```

Executing MonkeyRunner

To execute your MonkeyRunner script, have your device or emulator set up at a likely starting point (e.g., home screen), then execute the `monkeyrunner` command, passing it the path to your script (e.g., `monkeyrunner monkey_sample.py`). You will see the script executing on the screen of your device or emulator, and your console will contain whatever output you might emit from your test script itself. For example, you might take screenshots, compare them against a master copy (using methods on `MonkeyImage` to help with this), and emit warnings if they differ unexpectedly.

Monkeying Around

Independent from the JUnit system and MonkeyRunner is the Test Monkey (referred to here as "the Monkey" for short).

The Monkey is a test program that simulates random user input. It is designed for "bash testing", confirming that no matter what the user does, the application will not crash. The application may have odd results — random input entered into a Twitter client may, indeed, post that random input to Twitter. The Monkey does not test to make sure that results of random input make sense; it only tests to make sure random input does not blow up the program.

You can run the Monkey by setting up your initial starting point (e.g., the main activity in your application) on your device or emulator, then running a command like this:

```
adb shell monkey -p your.package.here -v --throttle 100 600
```

(substituting the package name of a project on your device or emulator for `your.package.here`)

Working from right to left, we are asking for 600 simulated events, throttled to run every 100 milliseconds. We want to see a list of the invoked events (`-v`) and we want to throw out any event that might cause the Monkey to leave our application, as determined by the application's package (`-p your.package.here`).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Monkey will simulate keypresses (both QWERTY and specialized hardware keys, like the volume controls), D-pad/trackball moves, and sliding the keyboard open or closed. Note that the latter may cause your emulator some confusion, as the emulator itself does not itself actually rotate, so you may end up with your screen appearing in landscape while the emulator is still, itself, portrait. Just rotate the emulator a couple of times (e.g., `<Ctrl>-<F12>`) to clear up the problem.

For playing with a Monkey, the above command works fine. However, if you want to regularly test your application this way, you may need some measure of repeatability. After all, the particular set of input events that trigger your crash may not come up all that often, and without that repeatable scenario, it will be difficult to repair the bug, let alone test that the repair worked.

To deal with this, the Monkey offers the `-s` switch, where you provide a seed for the random number generator. By default, the Monkey creates its own seed, giving totally random results. If you supply the seed, while the sequence of events is random, it is random for that seed — repeatedly using the same seed will give you the same events. If you can arrange to detect a crash and know what seed was used to create that crash, you may well be able to reproduce the crash.

There are many more Monkey options, to control the mix of event types, to generate profiling reports as tests are run, and so on. The Monkey documentation in the SDK's Developer's Guide covers all of that and more.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Testing with UIAutomator

Yet another approach for testing Android applications is `uiautomator`. This blends JUnit and Test Monkey approaches, offering some unique advantages over either of those individual frameworks.

However, at the present time, it would appear that the Android Studio beta does not have support for `uiautomator` projects. Hence, if you are not comfortable with Eclipse, this is not going to be a viable option for you right now.

Prerequisites

This chapter assumes that you have read the chapter on JUnit-based testing, mostly for comparison purposes. Having read the chapter on monkeyrunner and the Test Monkey is also a good idea, again for comparison purposes.

What Is UIAutomator?

`uiautomator`, as the name suggests, automates UIs. It simulates user input, in the form of tapping on items and the like. It does so without modifying your process' contents, and so in that respect it behaves like the Test Monkey.

However, unlike the Test Monkey, tests run by `uiautomator` are implemented in JUnit, and those tests have limited access to the widgets inside of a UI. Such access not only allows for directing simulated user input (e.g., "click the OK button"), but also for asserting that various test conditions are true (e.g., "does the list have five rows?"). In this respect, `uiautomator` behaves like traditional Android JUnit testing.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Why Choose UIAutomator Over Alternatives?

In some respects, `uiautomator` represents the worst of both worlds. You have to use JUnit, making test authoring a challenge for those not skilled with Java. Yet you only have fairly generic access to an activity's widgets, versus the complete white-box capability of normal instrumentation-based JUnit testing.

Hence, why would anyone bother?

The big thing that `uiautomator` offers over classic JUnit testing is greater ability to test an *application* versus testing *individual components*. The classic JUnit test cases are organized around testing some specific component, such as using `ActivityInstrumentationTestCase2` to exercise some specific activity. Testing the flow of work between activities is difficult from classic JUnit, but is relatively easy with `uiautomator`. You can also use this for integration testing, as you can exercise and analyze applications other than your own, such as to confirm that you are starting a third-party app correctly.

Similarly, classic JUnit testing cooks up activity instances "out of thin air". Instead, `uiautomator` executes normal UI operations to create the activities, such as tapping on your app's icon in the home screen launcher. This more accurately simulates what a user will do — users are far more likely to tap on a launcher than to hack into your Dalvik VM and manually instantiate an activity.

Creating Some Tests

As with JUnit, creating test cases for `uiautomator` involves creating a separate project. In this case, though, the project will create just an ordinary Java JAR file, not an APK, let alone one designed to run in the same process as your main app. `uiautomator` can then run that JAR to execute your tests, as we will see later in this chapter.

You can see a set of `uiautomator` tests in a suitable project in the `Testing/JUnit/uiautomator` directory. This is where the various code samples shown in this chapter come from.

Using this project straight from the GitHub repo will require some modifications, discussed below.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Setting Up for Command-Line Builds

While some pieces of the process for creating and executing `uiautomator` tests can be done from an IDE, not all can. Hence, you will need to be set up to support command-line builds in addition to any IDE you might already have configured.

Creating the Test Project

First, create an ordinary Java project, *not* an Android application project or an Android test project. You can place this project wherever you want on your development machine's filesystem — it does not have to be in some "magic directory" relative to the application being tested.

In that project, you will need to add two JARs to your classpath: `uiautomator.jar` and `android.jar`. The latter is the standard compile-time edition of the Android SDK, what you normally get in an Android application project by specifying your build target. The former, as the name might suggest, contains classes related to writing `uiautomator` tests.

These two JARs are Android version-specific, insofar as there are different editions of these JARs for each Android API level from API Level 16 on up. Since tests written using these JARs are backwards-compatible, you will typically choose the most recent edition of the JARs for your project. In particular, choosing a JAR that is for API Level 18 or higher is a good idea, to give you access to some newer automation APIs.

In Eclipse, you would add those JARs via right-clicking on the project, choosing Build Path > Configure Build Path from the context menu, switching to the Libraries tab, clicking "Add External JARs...", navigating to the appropriate `platforms/` subdirectory in your SDK installation (e.g., `platforms/android-17/`, and choosing those two JARs. Outside of Eclipse, you would add those two JARs to your compile-time classpath via whatever means are appropriate.

You also need to attach JUnit to the project. In Eclipse, to do this, you would right-click on the project, choose Build Path > Add Libraries..., choose JUnit from the resulting dialog, then choose JUnit3 from the final dialog. You would need to do something similar outside of Eclipse to attach JUnit to your compile-time classpath.

Then, you will need to make the project files necessary to prepare your code for use as a `uiautomator` test JAR. To do that, you will need to run the `android create uitest-project` command:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
android create uitest-project -n <name> -t android-17 -p <path>
```

where:

- `-n` is the name you wish to give the JAR, minus the `.jar` file extension. Note that the documentation presently claims that this is optional; it is not.
- `-t` to indicate what API level's JARs you are using. This needs to be the name of the directory from which you pulled the JARs (e.g., `android-17`). Note that the documentation presently shows something else (a value of `1`), which is incorrect.
- `-p [path>` needs to point to the directory in which your test project will reside (note: *not* your production project!)

If you are trying to use the sample test project from the GitHub repo, you will need to adjust the references to the external JARs (`android.jar` and `uiautomator.jar`) to point to your own SDK installation, as the project configuration files will be pointing elsewhere (unless your development environment is set up exactly the same as that of the author of this book, which would be really surprising).

Creating a Test Case

Your test case classes should inherit from `com.android.uiautomator.core.UiAutomatorTestCase`, supplied by the `uiautomator.jar` library. This will initialize the `uiautomator` connection to your device or emulator and give you access to it via a `UiDevice` object, retrieved by calling `getUiDevice()`. You will use this object as your primary gateway into the device for events and retrieving information about your UI, as will be seen in the next few sections.

Note that `UiAutomatorTestCase` itself inherits from the standard JUnit `TestCase` class, and so you are also welcome to create `setUp()` and `tearDown()` methods as you see fit.

Performing Device-Level Actions

The `UiDevice` you get from `getUiDevice()` has many methods that allow you to perform device-level actions, such as calling `pressHome()` to press the HOME button (and thereby bring up the home screen). Similarly, you can call:

- `pressBack()` and `pressMenu()` for the BACK and MENU buttons
- `pressDPadUp()`, `pressDPadLeft()`, etc. for D-pad events

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `pressRecentApps()` to bring up the recent tasks list
- `pressKeyCode()` to press an arbitrary key based on the keycode from `KeyEvent`

...and so on.

Inspecting and Interacting with the UI

Of course, pressing some buttons is not especially useful on its own, only as a means to an end, such as launching your activity. To do more than this, you will need to get your hands on widgets and containers, to perform operations related to them.

The key is that you can "get your hands on widgets and containers" *from whatever activity is in the foreground*. This is not limited to your own app, but rather works for *any* app, including the home screen itself.

The following sections will work through some common `uiautomator` operations, in the context of the `openActivity()` from the `ListTests` class in the sample project. This method, called from `setUp()`, consolidates the work to bring an instance of our production activity to the foreground, by means of interacting with the home screen:

```
  private void openActivity() throws UiObjectNotFoundException {
    getUiDevice().pressHome();

    UiObject allAppsButton=
        new UiObject(new UiSelector().description("Apps"));

    allAppsButton.clickAndWaitForNewWindow();

    UiObject appsTab=new UiObject(new UiSelector().text("Apps"));

    appsTab.click();

    UiScrollable appViews=
        new UiScrollable(new UiSelector().scrollable(true));

    appViews.setAsHorizontalList();

    UiObject ourApp=
        appViews.getChildByText(new
UiSelector().className("android.widget.TextView"),
                                "Action Bar Fragment Demo");

    ourApp.clickAndWaitForNewWindow();

    UiObject appValidation=
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        new UiObject(
                        new
UiSelector().packageName("com.commonsware.android.abf"));

    assertTrue("Could not open test app", appValidation.exists());
  }
```

Finding and Interaction with Widgets

`openActivity()` starts by calling `pressHome()` on the `UiDevice`, to ensure that the home screen is in the foreground:

```
    getUiDevice().pressHome();
```

Next, we want to bring up the home screen's launcher, showing the available launchable activities, so that we can find our app and launch it. What a user would do, on a stock Android environment like an emulator, would be to click on the appropriate button to bring up the launcher. We need to do the same thing, except from our test code. This implies:

- Finding that widget
- Simulating a click of that widget

Web developers are used to finding DOM nodes by CSS queries. Developers using XML are used to using XPath queries to find particular elements. Along the same lines, `uiautomator` gives us a flexible system to find widgets in the foreground activity, by means of a `UiSelector` object, typically created using the public zero-argument constructor (i.e., `new UiSelector()`).

In CSS, a "selector" can identify DOM nodes by class, `id`, or ones with particular properties. A `UiSelector` can do much the same thing. So, the first `UiSelector` created in `openActivity()` will find a widget in the foreground activity whose "description" is `Apps` (`new UiSelector().description("Apps")`). Here, "description" will mean either the text of a `TextView` or the `android:contentDescription` of other types of widgets.

How do we know that this particular button has a "description" of `Apps`? In this case, we found out using `uiautomatorviewer`, which will be discussed later in this chapter.

By passing our `UiSelector` to the constructor of `UiObject`, we get a `UiObject` that, hopefully, knows how to interact with this particular button of the home screen. In

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

particular, we call `clickAndWaitForNewWindow()` on it, which taps the button and blocks until something else (e.g., a new activity) has taken over the foreground:

```
    UiObject allAppsButton=
        new UiObject(new UiSelector().description("Apps"));

    allAppsButton.clickAndWaitForNewWindow();
```

The stock Android launcher has two tabs, one for apps and one for (app) widgets. We need to ensure that the apps tab is selected. So, once again, we create a `UiSelector` and use it to create a `UiObject` to represent the apps tab. This time, we use `text()` instead of `description()`. `text()` will find a widget based solely on its display text (e.g., `android:text` of a `TextView`). In truth, we could have used `description()` here as well, with the same results.

Then, we call `click()` on the `UiObject`, to simulate a tap on this tab, to ensure that is the selected tab.

```
    UiObject appsTab=new UiObject(new UiSelector().text("Apps"));

    appsTab.click();
```

Dealing with Collections

Finding widgets by text or description is fairly easy when there is only one possible widget that has that text or description. Things get more complicated when you are dealing with a collection of widgets, such as an `AdapterView`.

For example, the Apps tab of the standard Android launcher uses a `GridView` to show up to 20 launchable activities. Then, you need to swipe horizontally, courtesy of a `ViewPager`, to uncover additional `GridView` collections of launchable activities.

A `UiCollection` helps deal with this, in terms of allowing you to inspect a collection of widgets, including performing the necessary swipe operations to access all of the contents.

A `UiSelector` called with `scrollable(true)` will return a widget that is scrollable. Creating a `UiCollection` with that `UiSelector` will create a `UiCollection` around the first scrollable widget. In the case of the Apps tab, that will be the `ViewPager`-and-`GridView` combination.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In our case, to get to other elements in the collection, you need to swipe horizontally. To configure the `UiCollection` that way, we have to call `setAsHorizontalList()` on the `UiCollection`:

```
    UiScrollable appViews=
        new UiScrollable(new UiSelector().scrollable(true));

    appViews.setAsHorizontalList();
```

Finding Widgets By Type

In that collection, we want to find the item that contains our app's caption. This test project is designed to test the same sample app that was tested in the JUnit chapter, a slightly modified version of an early action bar sample. Our launcher entry's name will be "Action Bar Fragment Demo", as that is what we set up in the production project's manifest and string resources. So, we need to find the entry in the `ViewPager`-of-`GridViews` that has that title.

To do that, we will create yet another `UiSelector`. This time, though, we will find widgets by type, specifying `className("android.widget.TextView")` to only work with `TextView` widgets.

That `UiSelector` is passed into the `getChildByText()` method of `UiCollection`, which will iterate over the children to find the first one that matches the `UiSelector` and where the selected widget contains the supplied text:

```
    UiObject ourApp=
        appViews.getChildByText(new
UiSelector().className("android.widget.TextView"),
                                "Action Bar Fragment Demo");
```

Then, we again call `clickAndWaitForNewWindow()`, to tap on our launcher entry, triggering our app's activity to come to the foreground:

```
    ourApp.clickAndWaitForNewWindow();
```

Asserting Conditions

`UiSelector` and `UiObject` can also be used for some odd operations that do not fit the normal widgets-and-containers pattern shown above.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For example, now that we have opened a window from our app to be tested, it would be nice to confirm that, indeed, this is our app, and that our `openActivity()` method did not open some other app by mistake.

To do this, we can create a `UiSelector` and apply `packageName()`, to constrain the selection to widgets coming from an app with our desired package name:

```
    UiObject appValidation=
        new UiObject(
                        new
UiSelector().packageName("com.commonsware.android.abf"));
```

The `UiObject` we create always exists (i.e., is not `null`), as we are creating it via the constructor. However, it is entirely possible that our `UiSelector` cannot match any widget, such as would be the case if we accidentally opened the wrong app and tried to find a widget stemming from our package. The `exists()` method on a `UiObject` returns `true` if the `UiObject` is pointing at an actual widget, `false` otherwise. Hence, we can assert that we indeed have a widget coming from our package:

```
    assertTrue("Could not open test app", appValidation.exists());
```

The net result is that we open our main activity and confirm that, indeed, that is what we opened.

Two Sample Test Methods

All of that was just to get the activity for testing onto the screen.

Now the *real* testing begins.

The `ListTests` class has two test methods, `testContents()` and `testAdd()`, designed to (lightly) exercise the UI.

testContents()

The objective of the `testContents()` method is to confirm that the 25 words all appear in the `ListView`.

To do that, we:

- Create a `UiScrollable` for a `UiSelector` that finds the `ListView` in our activity

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Mark that `UiScrollable` as being a vertical list, where swipes up and down will expose the various children
- Iterate over the array of words, finding the `TextView` for each word and confirming that this widget does indeed exist

```
  public void testContents() throws UiObjectNotFoundException {
    UiScrollable words=
        new UiScrollable(
                        new UiSelector().className("android.widget.ListView"));

    words.setAsVerticalList();

    for (String s : items) {
      assertNotNull("Could not find " + s,
                    words.getChildByText(new
UiSelector().className("android.widget.TextView"),
                                         s));
    }
  }
```

testAdd()

The objective of the `testAdd()` method is to add a new word to the list, via the `EditText` widget in our action bar, then confirm that the new word was actually added to the list.

To do that, we:

- Retrieve the `EditText` by finding the widget whose text is "Word" (the hint of our `EditText`)
- Call `setText()` to fill in `snicklefritz` into the `EditText` widget, which `UiObject` accomplishes by actually typing in the value
- Call `pressEnter()` on the `UiDevice` to simulate pressing the Enter key of a keyboard, which will trigger our action listener in the test activity and will add the word to the list
- Create a `UiScrollable` for a `UiSelector` that finds the `ListView` in our activity
- Mark that `UiScrollable` as being a vertical list, where swipes up and down will expose the various children
- Try to find a `TextView` whose text is `snicklefritz` and assert that it was found

```
  public void testAdd() throws UiObjectNotFoundException {
    UiObject add=new UiObject(new UiSelector().text("Word"));
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    add.setText("snicklefritz");
    getUiDevice().pressEnter();

    UiScrollable words=
        new UiScrollable(
                        new UiSelector().className("android.widget.ListView"));

    words.setAsVerticalList();

    assertNotNull("Could not find snicklefritz",
                  words.getChildByText(new
UiSelector().className("android.widget.TextView"),
                                       "snicklefritz"));
  }
```

Cleaning Up

Our `ListTests` class also has a `tearDown()` method, invoked by JUnit after each test method. Here, we press BACK twice, to return us to the main home screen from our activity, setting things back up for the next test method:

```
  @Override
  public void tearDown() {
    getUiDevice().pressBack();
    getUiDevice().pressBack();
  }
```

Running Your Tests

At the present time, running your tests is a fairly manual process, driven from the command line. In the future, with luck, there will be better tooling to assist with this.

Building and Pushing the JAR

Your test project, created via `android create uitest-project`, should contain an Ant `build.xml` file. Hence, running `ant build` from the root of your test project's directory should create a JAR file in `bin/` containing your compiled test code.

You then need to push that to the device's temporary directory, `/data/local/tmp`, such as via the `adb push` command:

```
adb push bin/NameOfYour.jar /data/local/tmp/
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Executing uiautomator

To actually run the tests, you will use `adb shell` to run the `uiautomator` command on the connected device or emulator:

```
adb shell uiautomator runtest <jar> -c <tests>
```

where:

- `<jar>` is the name of the test JAR that you pushed to the device in the preceding step.
- `<tests>` is the fully-qualified name of your test class (e.g., `com.commonsware.android.abf.uiautomator.ListTests`), optionally with a suffix identifying a single test method to run (e.g., `com.commonsware.android.abf.uiautomator.ListTests#testContents`. Without this suffix, all test methods on the class will be run. You can specify two or more classes (or class+methods) using a space as a separator between them.

There are other command-line switches that you can optionally provide to further customize the behavior of your tests, though the ones listed above are all that are needed.

As with running a classic set of JUnit tests, the output of the test run is dumped to your terminal window, showing you the successes and failures.

Note that the device or emulator will have to be set up with your production app, so that your test code that launches it will succeed. Since the test project and the production project are decoupled, simply running the `uiautomator` does not automatically push a new copy of your app to the device or emulator, though you could augment some build scripts or a wrapper shell script/batch file to do this for you.

Finding Your Widgets

The key to finding your desired widgets stems in large part from the `text()` or `description()` methods on `UiSelector`. Of those two, the latter is more flexible, as it will use the `android:contentDescription` from any widget, while `text()` is limited to `TextView` and its subclasses.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, this implies that your widgets have `android:contentDescription` defined. This is also important for accessibility, and therefore is a good idea regardless of its use with `uiautomator`.

For testing your own code, you can also find widgets via their resource IDs. `UiSelector` has `resourceId()` and `resourceIdMatches()` methods to configure the resource ID you want. As the `resourceIdMatches()` method name suggests, the resource ID here is a *string representation* of the resource name. It will be of the form `your.app.package:id/resource` (e.g., `com.commonsware.android.hotkey:id/editor`). However:

- Note that this requires API Level 18 or higher versions of the two JARs (`uiautomator.jar` and `android.jar`)
- Note that this requires running the tests on an API Level 18+ device or emulator
- Bear in mind that third party apps are welcome to rename their widgets when they wish, so your integration tests may break when third parties do so

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Advanced UI

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Introducing GridLayout

In 2011, Google added `GridLayout` to our roster of available container classes (a.k.a., layout managers). `GridLayout` is an attempt to make setting up complex Android layouts a bit easier, particularly with an eye towards working well with IDE graphical layout editors. In this chapter, we will examine why `GridLayout` was added and how we can use it in our projects.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Issues with the Classic Containers

Most layouts are implemented using a combination of `LinearLayout`, `RelativeLayout`, and `TableLayout`. Almost everything you would want to be able to create can be accomplished using one, or sometimes more than one, of those containers.

However, there are issues with the classic containers. The two most prominent might be the over-reliance upon nested containers and issues with drag-and-drop GUI building capability.

Nested Containers

`LinearLayout` and `TableLayout` suffer from a tendency to put too many containers inside of other containers. For example, implementing some sort of 2x2 grid would involve:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- A vertical `LinearLayout` holding onto a pair of horizontal `LinearLayouts`, or
- A `TableLayout` holding onto a pair of `TableRows`

On the surface, this does not seem that bad. And, in many cases, it is not that bad.

However, views and containers are relatively heavyweight items. They consume a fair bit of heap space, and when it comes time to lay them out on the screen, they consume a fair bit of processing power. In particular, the fact that a container can hold onto *any* type of widget or container means that it is difficult to optimize common scenarios (e.g., a 2x2 grid) for faster processing. Instead, a container treats its children more or less as "black boxes", requiring lots of method invocations up and down the call stack to calculate sizes and complete the layout process.

Moreover, the call stack itself can be an issue. The stack size of the main application thread has historically been rather small (8KB was the last reported value). If you have a complex UI, with more than ~15 nested containers, you are likely to run into a `StackOverflowError`. Android itself will contribute some of these containers, exacerbating this problem.

`RelativeLayout`, by comparison, can implement some UI patterns without any nested containers, simply by positioning widgets relative to the container's bounds and relative to each other.

Drag-and-Drop

Where `RelativeLayout` falls down is with the drag-and-drop capability of the graphical layout editor in IDEs like Eclipse and Android Studio.

When you release the mouse button when dropping a widget into the preview area, the tools need to determine what that really means in terms of layout rules.

`LinearLayout` works fairly well: it will either insert your widget in between two other widgets or add it to the end of the row or column you dropped into. `TableLayout` behaves similarly.

`RelativeLayout`, though, has a more difficult time guessing what particular combination of rules you really mean by this particular drop target. Are you trying to attach the widget to another widget? If so, which one? Are you trying to attach the widget to the bounds of the `RelativeLayout`? While sometimes it will guess properly, sometimes it will not, with potentially confusing results. It is reasonably

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

likely that you will need to tweak the layout rules manually, either via the Properties pane or via the raw XML.

The New Contender: GridLayout

`GridLayout` tries to cull the best of the capabilities of the classic containers and drop as many of their limitations as possible.

`GridLayout` works a bit like `TableLayout`, insofar as it sets things up in a grid, with rows and columns, where the row and column sizes are computed based upon what is placed into those rows and columns. However, unlike `TableLayout`, which relies upon a separate `TableRow` container to manage the rows, `GridLayout` takes the `RelativeLayout` approach of putting rules on the individual widgets (or containers) in the grid, where those rules steer the layout processing. For example, with `GridLayout`, widgets can declare specifically which row and column they should slot into.

`GridLayout` also goes a bit beyond what `TableLayout` offers in terms of capabilities. Notably, it supports row spans as well as column spans, whereas `TableRow` only supports a column span. This gives you greater flexibility when designing your layout to fit the grid-style positioning rules. You can also:

- Explicitly state how many columns there are, rather than having that value be inferred by row contents
- Allow Android to determine where to place a widget without specifying any row or column, with it finding the next available set of grid cells capable of holding the widget, based upon its requested row span and column span values
- Have control over orientation: whereas `TableLayout` always was a column of rows, you could have a `GridLayout` be a row of columns, if that makes implementing the design easier
- And so on

GridLayout and the Android Support Package

`GridLayout` was natively added to the Android SDK in API Level 14 (Android 4.0). Fortunately, the Android Support package has a backport of `GridLayout`. However, the backport is not in one of the JAR files, such as `support-v4`, as `GridLayout` requires some resources. Hence, it is in an Android library project that you must add to your project, known as `gridlayout-v7`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio users can simply add a `compile 'com.android.support:gridlayout-v7:...'` statement to their top-level `dependencies` closure, for some version identified by `...`. So long as those users have the Android Repository set up in the SDK Manager, Gradle will be able to find and incorporate the artifact.

Eclipse users can copy the library project from `$SDK/extras/android/support/v7/gridlayout` (where `$SDK` is wherever you installed your copy of the Android SDK) to some other spot on their development machine. Then, they can import the library project and attach it to their app project, as was described in the chapter on library projects.

When using the backported `GridLayout`, you will need to declare another XML namespace in your layout XML resources. That namespace will be `http://schemas.android.com/apk/res-auto`. If you use an IDE to add the `GridLayout` to the layout resource, it will automatically add this namespace, under the prefix of `app`, such as:

```
<android.support.v7.widget.GridLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:columnCount="2">
</android.support.v7.widget.GridLayout>
```

That namespace is required for `GridLayout`-specific attributes. For example, we can have a `columnCount` attribute, indicating how many columns the `GridLayout` should contain. For the native API Level 14 `GridLayout`, that attribute would be `android:columnCount`. For the backport, it will be `app:columnCount`, assuming that you gave the namespace the prefix of `app`.

When citing `GridLayout`-specific attributes, the rest of this chapter will use the `app` prefix, to clarify which attributes need that prefix for the backport. If you are using the native API Level 14 implementation of `GridLayout`, and you are manually working with the XML, just remember to use `android` as a prefix instead of `app`.

The sample app shows *both* the native and the backport implementations of `GridLayout`: on API Level 14+ devices/emulators it will use native implementations from `res/layout-v14/`, and it will use the backport on older environments.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Eclipse and GridLayout

You will find `GridLayout` in the "Layouts" portion of the palette of available widgets and containers. As with anything else, you can drag it from the palette into the preview area, then use the Outline and Properties views to configure it.

However, this is the native API Level 14 version of `GridLayout`, not the backport. If you wish to use the backport, you will need to go into the XML and manually adjust the element name, to be `android.support.v7.widget.GridLayout` instead of `GridLayout`. This may, in turn, require restarting Eclipse to make it happy, if you get errors in the preview area when trying to view the resulting `GridLayout`.

Trying to Have Some Rhythm

One of the things that the Android design guidelines try to emphasize is having everything work in `48dp`-high blocks, to give you a reasonable set of touch targets for fingers, while maintaining some uniformity of sizing.

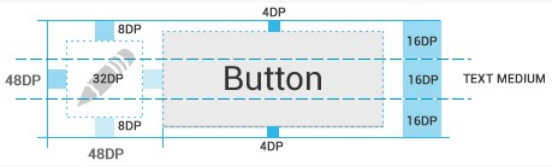


Figure 357: 48dp Rhythm, Depicted (image courtesy of Android Open Source Project)

When working with `GridLayout` in the Eclipse graphical layout editor, you will be prompted to try to put your widgets in `48dp`-based positions. There is even a "snap to grid" toolbar toggle button that, when pressed, will force everything you drag into the `GridLayout` to reside on a `16dp`-based grid.

To accomplish this, it adds a series of `Space` widgets to your layout (or, if you are working with the backport of `GridLayout`, you get `android.support.v7.widget.Space` widgets). These will consume the space that gets your widgets to line up in what Eclipse thinks is the proper positioning. We will see examples of that as we examine our sample application.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Our Test App

To look at a series of GridLayout-based layouts, let's turn our attention to the GridLayout/Sampler sample project. This has the same ViewPager and PagerTabStrip as did the second sample app from the chapter on ViewPager. However, rather than use a list of 10 EditText widgets managed by fragments, in this case, our fragments will manage layouts containing GridLayout. Each page of our pager will contain a TrivialFragment, whose contents are based on a Sample class that is a simple pair of a layout resource ID and a string resource ID for the fragment's title:

```
package com.commonsware.android.gridlayout;

class Sample {
  int layoutId;
  int titleId;

  Sample(int layoutId, int titleId) {
    this.layoutId=layoutId;
    this.titleId=titleId;
  }
}
```

Our revised SampleAdapter maintains a static ArrayList of these Sample objects, one per layout we wish to examine, and uses those values to populate our ViewPager title:

```
package com.commonsware.android.gridlayout;

import android.app.Fragment;
import android.app.FragmentManager;
import android.content.Context;
import android.support.v13.app.FragmentPagerAdapter;
import java.util.ArrayList;

public class SampleAdapter extends FragmentPagerAdapter {
  static ArrayList<Sample> SAMPLES=new ArrayList<Sample>();
  private Context ctxt=null;

  static {
    SAMPLES.add(new Sample(R.layout.row, R.string.row));
    SAMPLES.add(new Sample(R.layout.column, R.string.column));
    SAMPLES.add(new Sample(R.layout.table, R.string.table));
    SAMPLES.add(new Sample(R.layout.table_flex, R.string.flexible_table));
    SAMPLES.add(new Sample(R.layout.implicit, R.string.implicit));
    SAMPLES.add(new Sample(R.layout.spans, R.string.spans));
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  public SampleAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);
    this.ctxt=ctxt;
  }

  @Override
  public int getCount() {
    return(SAMPLES.size());
  }

  @Override
  public Fragment getItem(int position) {
    return(TrivialFragment.newInstance(getSample(position).layoutId));
  }

  @Override
  public String getPageTitle(int position) {
    return(ctxt.getString(getSample(position).titleId));
  }

  private Sample getSample(int position) {
    return(SAMPLES.get(position));
  }
}
```

`TrivialFragment` just inflates our desired layout, having received the layout resource ID as a parameter to its factory method:

```
package com.commonsware.android.gridlayout;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class TrivialFragment extends Fragment {
  private static final String KEY_LAYOUT_ID="layoutId";

  static TrivialFragment newInstance(int layoutId) {
    TrivialFragment frag=new TrivialFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_LAYOUT_ID, layoutId);
    frag.setArguments(args);

    return(frag);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(inflater.inflate(getArguments().getInt(KEY_LAYOUT_ID, -1),
                            container, false));
  }
}
```

Note that if you load this project from the GitHub repository, you will need to update it for your copy of the `GridLayout` library project.

Replacing the Classics

Let's first examine the behavior of `GridLayout` by seeing how it can replace some of the classic layouts we would get from `LinearLayout` and `TableLayout`. Each of the following sub-sections will examine one `GridLayout`-based layout XML resource, how it can be constructed, and what the result looks like when viewed in the sample project.

Horizontal LinearLayout

The classic way to create a row of widgets is to use a horizontal `LinearLayout`. The `LinearLayout` will put each of its children, one after the next, within the row.

The `GridLayout` equivalent is to specify one that has an `app:columnCount` equal to the number of widgets in the row. Then, each widget will have `app:layout_column` set to its specific column index (starting at `0`) and `app:layout_row` set to `0`, as seen in `res/layout/row.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/
apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:columnCount="2">

  <Button
    app:layout_column="0"
    app:layout_row="0"
    android:text="@string/button"/>

  <Button
    app:layout_column="1"
    app:layout_row="0"
    android:text="@string/button"/>

</android.support.v7.widget.GridLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Unlike `LinearLayout`, though, we do not specify sizes of the children, in terms of `android:layout_width` and `android:layout_height`. `GridLayout` works a bit like `TableLayout` in this regard, supplying default values for these attributes. In the case of `GridLayout`, the defaults are `wrap_content`, and this cannot be overridden (akin to the behavior of immediate children of a `TableRow`). Instead, you will control size via row and column spans, as will be illustrated later in this chapter.

Given the above layout, we get:

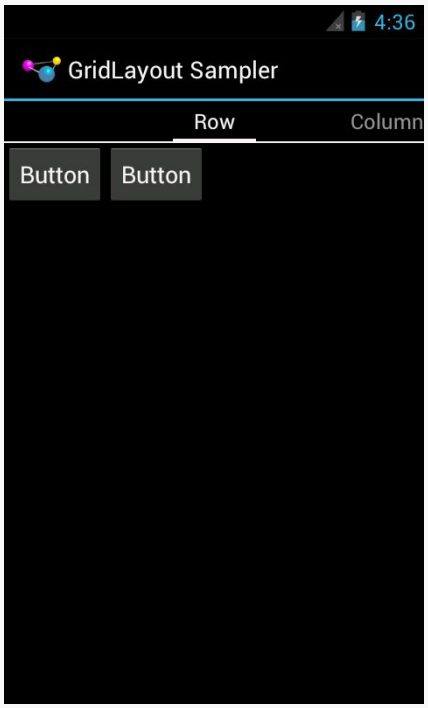


Figure 358: Row Using GridLayout, on a 4.0.3 Emulator

Vertical LinearLayout

Similarly, the conventional way you would specify a column is to use a vertical `LinearLayout`, which would position its children one after the next. The `GridLayout` equivalent would be to have `app:columnCount` set to `1`, and to place the widgets in each required row via `app:layout_row` attributes, as seen in `res/layout/column.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/
apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:layout_height="match_parent"
  app:columnCount="1">

  <Button
    app:layout_column="0"
    app:layout_row="0"
    android:text="@string/button"/>

  <Button
    app:layout_column="0"
    app:layout_row="1"
    android:text="@string/button"/>

</android.support.v7.widget.GridLayout>
```

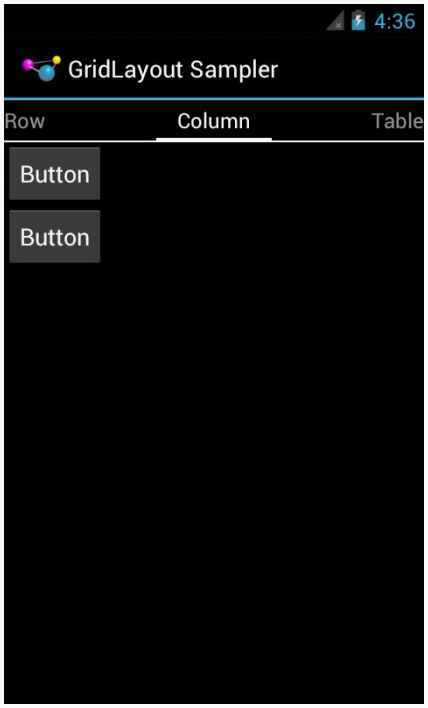


Figure 359: Column Using GridLayout, on a 4.0.3 Emulator

All that being said, it is still probably better to use `LinearLayout` in these cases, rather than mess with `GridLayout`.

TableLayout

The big key to a `TableLayout` is column width, where columns expand to fill their contents, assuming there is sufficient room in the table. `GridLayout` also expands its columns to address the sizes of its contents.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For example, here is a simple 2x2 table, with `TextView` widgets in the left column and `EditText` widgets in the right column, as seen in `res/layout/table.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/
apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:columnCount="2">

  <TextView
    app:layout_column="0"
    app:layout_row="0"
    android:text="@string/name"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_column="1"
    app:layout_row="0"
    android:inputType="textPersonName">

    <requestFocus/>
  </EditText>

  <TextView
    app:layout_column="0"
    app:layout_row="1"
    android:text="@string/address"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_column="1"
    app:layout_row="1"
    android:inputType="textPostalAddress"/>

</android.support.v7.widget.GridLayout>
```

One feature of the Eclipse graphical layout editor is that we can toggle on a series of lines showing the sizing of the rows and columns, by clicking the "Show Structure" toolbar button:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 360: "Show Structure" Toolbar Icon

This helps illustrate that our right column actually takes up all remaining room on the screen, by showing green gridlines denoting the transitions between rows and columns:

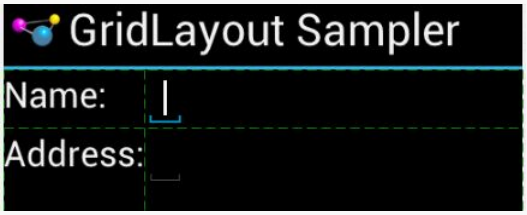


Figure 361: "Show Structure" Output for GridLayout Table Layout

However, our `EditText` widgets are small, because nothing is causing them to fill the available space. To do that, we can use `android:layout_gravity`, to ask the `GridLayout` to let the widgets fill the available horizontal space, as seen in `res/layout/table_flex.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/
apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:columnCount="2">

  <TextView
    app:layout_column="0"
    app:layout_row="0"
    android:text="@string/name"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    app:layout_column="1"
    app:layout_row="0"
    app:layout_gravity="fill_horizontal"
    android:inputType="textPersonName">

    <requestFocus/>
  </EditText>

  <TextView
    app:layout_column="0"
    app:layout_row="1"
    android:text="@string/address"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_column="1"
    app:layout_row="1"
    app:layout_gravity="fill_horizontal"
    android:inputType="textPostalAddress"/>

</android.support.v7.widget.GridLayout>
```

This allows the `EditText` widgets to fill the width of the column:

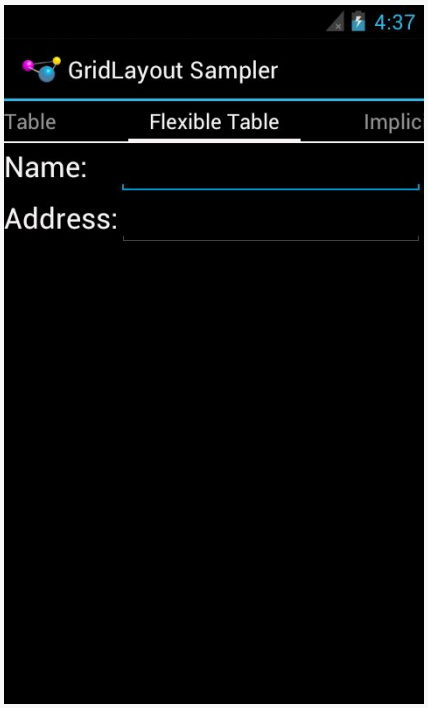


Figure 362: Table Using GridLayout, on a 4.0.3 Emulator

That holds true regardless of how wide that column is:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 363: Table Using GridLayout, in Landscape, on a 4.0.3 Emulator

Implicit Rows and Columns

While all the previous samples showed the row and column of each widget being defined explicitly via `app:layout_row` and `app:layout_column` attributes, that is not your only option.

If you have `app:columnCount` on the `GridLayout` element itself, you can allow `GridLayout` to assign rows and columns. In this respect, `GridLayout` behaves a bit like a "flow layout": it assigns widgets to cells in the first row, starting from the first column and working its way across, wrapping to the next row when it runs out of room. This makes for a more terse layout file, at the cost of perhaps introducing a bit of confusion when you add or remove a widget and everything after it in the layout file shifts location.

For example, `res/layout/implicit.xml` is the same as `res/layout/table_flex.xml`, except that it skips the `app:layout_row` and `app:layout_column` attributes, allowing `GridLayout` to assign the positions:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/
apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:columnCount="2"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  app:orientation="horizontal">

  <TextView
    android:text="@string/name"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_gravity="fill_horizontal"
    android:inputType="textPersonName">

    <requestFocus/>
  </EditText>

  <TextView
    android:text="@string/address"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_gravity="fill_horizontal"
    android:inputType="textPostalAddress"/>

</android.support.v7.widget.GridLayout>
```

Visually, this sample is identical to the last one:

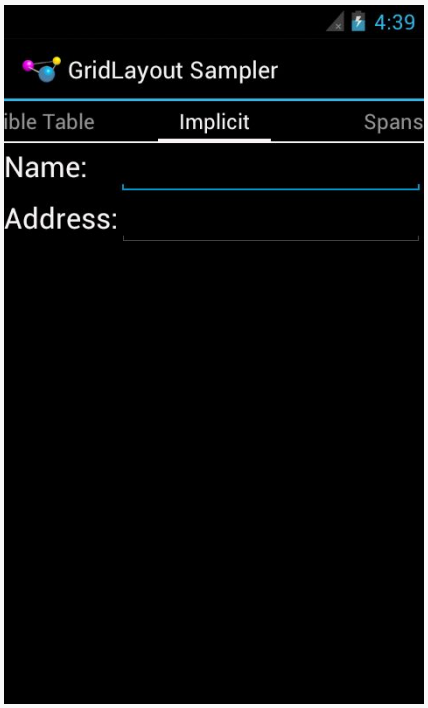


Figure 364: Table Using GridLayout and Implicit Positions, on a 4.0.3 Emulator

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The “across columns, then down rows” model holds for `GridLayout` in the default orientation: horizontal. You can add an `app:orientation` attribute to the `GridLayout`, setting it to `vertical`. Then, based on an `app:rowCount` value, `GridLayout` will automatically assign positions, working down the first column, then across to the next column when it runs out of rows.

Row and Column Spans

Like `TableLayout`, `GridLayout` supports the notion of column spans. You can use `app:layout_columnSpan` to indicate how many columns a particular widget should span in the resulting grid.

However, `GridLayout` also supports row spans, in the form of `app:layout_rowSpan` attributes. A widget can span rows, columns, or both, as needed.

If you are using implicit positions, per the previous section, `GridLayout` will seek the next available space that has sufficient rows and columns for a widget’s set of spans.

For example, the following diagram depicts five buttons placed in a `GridLayout` with various spans, and an attempt to add a sixth button that should span two columns:

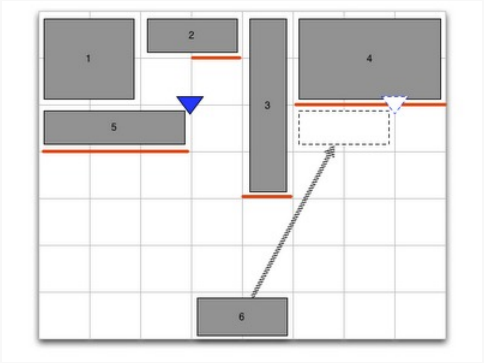


Figure 365: Span Sample (image courtesy of Android Open Source Project)

Assuming the first five buttons were added in sequence and with implicit positioning, `GridLayout` ordinarily would drop the sixth button into the fourth column of the third row. However, there is only a one-column-wide space available there, given that the third button intrudes into the third row. Hence, `GridLayout` will skip over the smaller space and put the sixth button into the sixth column in the third row.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A `GridLayout`-based layout that implements the above diagram can be found in `res/layout/spans.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:columnCount="9"
  app:orientation="horizontal"
  app:rowCount="5">

  <Button
    app:layout_gravity="fill"
    app:layout_columnSpan="2"
    app:layout_rowSpan="2"
    android:text="@string/string_1" />

  <Button
    app:layout_gravity="fill_horizontal"
    app:layout_columnSpan="2"
    android:text="@string/string_2" />

  <Button
    app:layout_gravity="fill_vertical"
    app:layout_rowSpan="4"
    android:text="@string/string_3" />

  <Button
    app:layout_gravity="fill"
    app:layout_columnSpan="3"
    app:layout_rowSpan="2"
    android:text="@string/string_4" />

  <Button
    app:layout_gravity="fill_horizontal"
    app:layout_columnSpan="3"
    android:text="@string/string_5" />

  <Button
    app:layout_gravity="fill_horizontal"
    app:layout_columnSpan="2"
    android:text="@string/string_6" />

  <android.support.v7.widget.Space
    android:layout_width="36dp"
    app:layout_column="0"
    app:layout_row="4" />

  <android.support.v7.widget.Space
    android:layout_width="36dp"
    app:layout_column="1"
    app:layout_row="4" />
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="2"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="3"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="4"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="5"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="6"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_width="36dp"
  app:layout_column="7"
  app:layout_row="4"/>

<android.support.v7.widget.Space
  android:layout_height="36dp"
  app:layout_column="8"
  app:layout_row="0"/>

<android.support.v7.widget.Space
  android:layout_height="36dp"
  app:layout_column="8"
  app:layout_row="1"/>

<android.support.v7.widget.Space
  android:layout_height="36dp"
  app:layout_column="8"
  app:layout_row="2"/>

<android.support.v7.widget.Space
  android:layout_height="36dp"
  app:layout_column="8"
  app:layout_row="3"/>

<android.support.v7.widget.Space
  android:layout_height="36dp"
  app:layout_column="8"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    app:layout_row="4"/>

</android.support.v7.widget.GridLayout>
```

This layout shows one of the limitations of `GridLayout`: its columns and rows will have a size of `0` by default. Hence, to ensure that each row and column has a minimum size, this layout uses `Space` elements (in an eighth column and fifth row) to establish those minimums. This makes the layout file fairly verbose, but it gives the desired results:

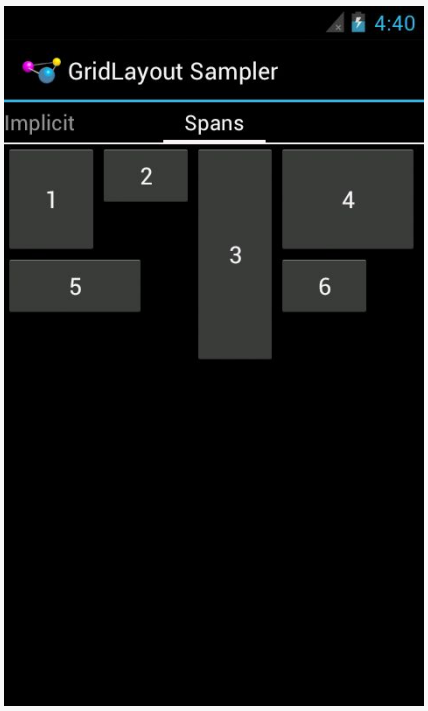


Figure 366: GridLayout Spans, on a 4.0.3 Emulator

However, the fixed-sized `Space` elements break the fluidity of the layout:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

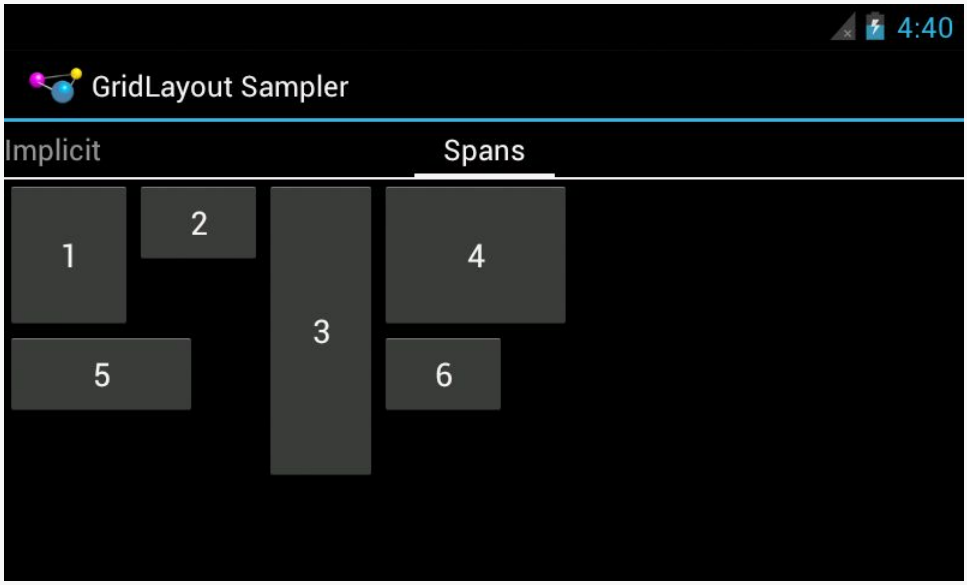


Figure 367: GridLayout Spans, in Landscape, on a 4.0.3 Emulator

Perhaps someday someone will create a `PercentSpace` widget, occupying a percentage of the parent's size, that could be used instead.

The author would like to give thanks to those on Stack Overflow who assisted in getting the span layout to work.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Dialogs and DialogFragments

Generally speaking, modal dialogs are considered to offer poor UX, particularly on mobile devices. You want to give the user more choices, not fewer, and so locking them into "deal with this dialog right now, or else" is not especially friendly. That being said, from time to time, there will be cases where that sort of modal interface is necessary, and to help with that, Android does have a dialog framework that you can use.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

DatePickerDialog and TimePickerDialog

Android has a pair of built-in dialogs that handle the common operations of allowing the user to select a date (`DatePickerDialog`) or a time (`TimePickerDialog`). These are simply dialog wrappers around the `DatePicker` and `TimePicker` widgets, as are described in this book's Widget Catalog.

The `DatePickerDialog` allows you to set the starting date for the selection, in the form of a year, month, and day of month value. Note that the month runs from `0` for January through `11` for December. Most importantly, both let you provide a callback object (`OnDateChangedListener` or `OnDateSetListener`) where you are informed of a new date selected by the user. It is up to you to store that date someplace, particularly if you are using the dialog, since there is no other way for you to get at the chosen date later on.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Similarly, `TimePickerDialog` lets you:

- Set the initial time the user can adjust, in the form of an hour (`0` through `23`) and a minute (`0` through `59`)
- Indicate if the selection should be in 12-hour mode with an AM/PM toggle, or in 24-hour mode (what in the US is thought of as "military time" and what in much of the rest of the world is thought of as "the way times are supposed to be")
- Provide a callback object (`OnTimeChangedListener` or `OnTimeSetListener`) to be notified of when the user has chosen a new time, which is supplied to you in the form of an hour and minute

For example, from the Dialogs/Chrono sample project, here's a trivial layout containing a label and two buttons — the buttons will pop up the dialog flavors of the date and time pickers:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TextView android:id="@+id/dateAndTime"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    />
  <Button android:id="@+id/dateBtn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Set the Date"
    android:onClick="chooseDate"
    />
  <Button android:id="@+id/timeBtn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Set the Time"
    android:onClick="chooseTime"
    />
</LinearLayout>
```

The more interesting stuff comes in the Java source:

```
package com.commonsware.android.chrono;

import android.app.Activity;
import android.app.DatePickerDialog;
import android.app.TimePickerDialog;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.os.Bundle;
import android.text.format.DateUtils;
import android.view.View;
import android.widget.DatePicker;
import android.widget.TextView;
import android.widget.TimePicker;
import java.util.Calendar;

public class ChronoDemo extends Activity {
  TextView dateAndTimeLabel;
  Calendar dateAndTime=Calendar.getInstance();

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    dateAndTimeLabel=(TextView)findViewById(R.id.dateAndTime);

    updateLabel();
  }

  public void chooseDate(View v) {
    new DatePickerDialog(this, d,
                         dateAndTime.get(Calendar.YEAR),
                         dateAndTime.get(Calendar.MONTH),
                         dateAndTime.get(Calendar.DAY_OF_MONTH))
      .show();
  }

  public void chooseTime(View v) {
    new TimePickerDialog(this, t,
                         dateAndTime.get(Calendar.HOUR_OF_DAY),
                         dateAndTime.get(Calendar.MINUTE),
                         true)
      .show();
  }

  private void updateLabel() {
    dateAndTimeLabel
      .setText(DateUtils
                 .formatDateTime(this,
                                 dateAndTime.getTimeInMillis(),

DateUtils.FORMAT_SHOW_DATE|DateUtils.FORMAT_SHOW_TIME));
  }

  DatePickerDialog.OnDateSetListener d=new DatePickerDialog.OnDateSetListener() {
    public void onDateSet(DatePicker view, int year, int monthOfYear,
                          int dayOfMonth) {
      dateAndTime.set(Calendar.YEAR, year);
      dateAndTime.set(Calendar.MONTH, monthOfYear);
      dateAndTime.set(Calendar.DAY_OF_MONTH, dayOfMonth);
      updateLabel();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  };

  TimePickerDialog.OnTimeSetListener t=new TimePickerDialog.OnTimeSetListener() {
    public void onTimeSet(TimePicker view, int hourOfDay,
                          int minute) {
      dateAndTime.set(Calendar.HOUR_OF_DAY, hourOfDay);
      dateAndTime.set(Calendar.MINUTE, minute);
      updateLabel();
    }
  };
}
```

The "model" for this activity is just a `Calendar` instance, initially set to be the current date and time. In the `updateLabel()` method, we take the current `Calendar`, format it using `DateUtils` and `formatDateTime()`, and put it in the `TextView`. The nice thing about using Android's `DateUtils` class is that it will format dates and times using the user's choice of date formatting, determined through the Settings application.

Each button has a corresponding method that will get control when the user clicks it (`chooseDate()` and `chooseTime()`). When the button is clicked, either a `DatePickerDialog` or a `TimePickerDialog` is shown. In the case of the `DatePickerDialog`, we give it an `OnDateSetListener` callback that updates the `Calendar` with the new date (year, month, day of month). We also give the dialog the last-selected date, getting the values out of the `Calendar`. In the case of the `TimePickerDialog`, it gets an `OnTimeSetListener` callback to update the time portion of the `Calendar`, the last-selected time, and a true indicating we want 24-hour mode on the time selector

With all this wired together, the resulting activity looks like this:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

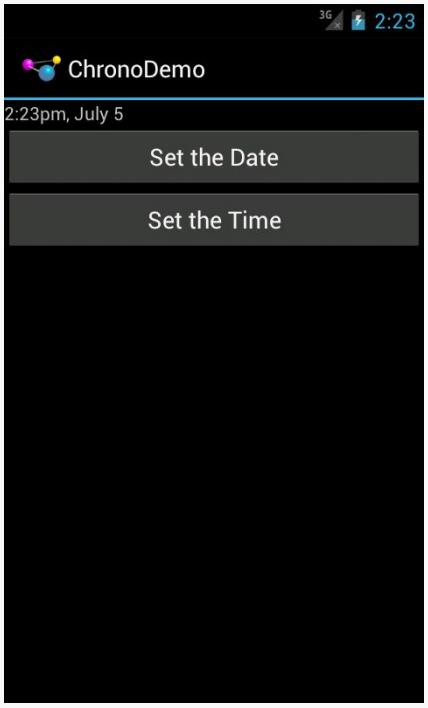


Figure 368: ChronoDemo, As Initially Launched, on Android 4.0.3

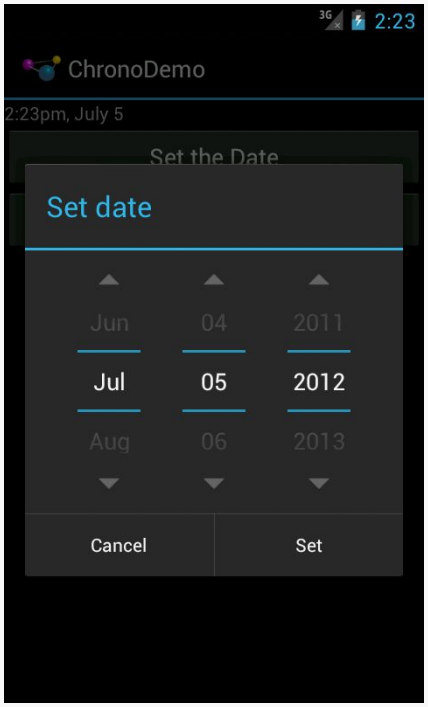


Figure 369: ChronoDemo, Showing DatePickerDialog

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

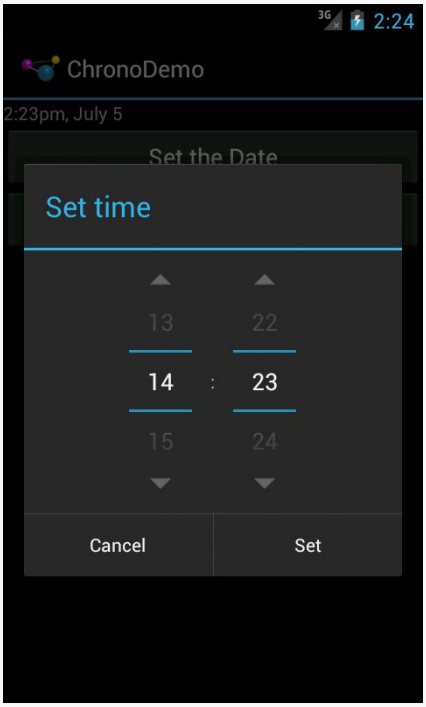


Figure 370: ChronoDemo, Showing TimePickerDialog

Changes (and Bugs) in Jelly Bean

`DatePickerDialog` and `TimePickerDialog` were modified in Android 4.1, and not necessarily for the better.

First, the "Cancel" button has been removed, unless you specifically add a negative button listener to the underlying `DatePicker` or `TimePicker` widget:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

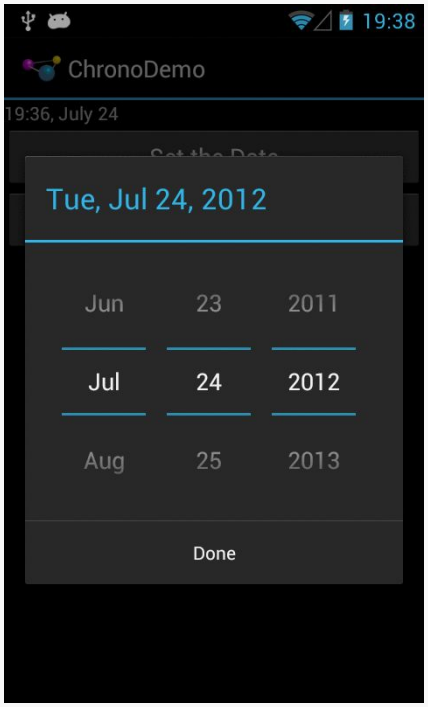


Figure 371: ChronoDemo, Showing DatePickerDialog, on a Jelly Bean Nexus S

The user can press BACK to exit the dialog, so all functionality is still there, but you may need to craft your documentation to accommodate this difference.

Then, your `OnDateSetListener` or `OnTimeSetListener` will be called an extra time. If the user presses BACK to leave the dialog, your `onDateSet()` or `onTimeSet()` will be called. If the user clicks the positive button of the dialog, you are called *twice*. There is a workaround documented on Stack Overflow, and the bug report can be found on the Android issue tracker.

AlertDialog

For your own custom dialogs, you could extend the `Dialog` base class, as do `DatePickerDialog` and `TimePickerDialog`. More commonly, though, developers create custom dialogs via `AlertDialog`, in large part due to the existence of `AlertDialog.Builder`. This builder class allows you to construct a custom dialog using a single (albeit long) Java statement, rather than having to create your own custom subclass. `Builder` offers a series of methods to configure an `AlertDialog`, each method returning the `Builder` for easy chaining.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Commonly-used configuration methods on Builder include:

- `setMessage()` if you want the "body" of the dialog to be a simple textual message, from either a supplied String or a supplied string resource ID.
- `setTitle()` and `setIcon()`, to configure the text and/or icon to appear in the title bar of the dialog box.
- `setPositiveButton()`, `setNeutralButton()`, and `setNegativeButton()`, to indicate which button(s) should appear across the bottom of the dialog, where they should be positioned (left, center, or right, respectively), what their captions should be, and what logic should be invoked when the button is clicked (besides dismissing the dialog).

Calling `create()` on the `Builder` will give you the `AlertDialog`, built according to your specifications. You can use additional methods on `AlertDialog` itself to perhaps configure things beyond what `Builder` happens to support.

Note, though, that calling `create()` does not actually display the dialog. The modern way to display the dialog is to tie it to a `DialogFragment`, as will be discussed in the next section.

DialogFragments

One challenge with dialogs comes with configuration changes, notably screen rotations. If they pivot the device from portrait to landscape (or vice versa), presumably the dialog should remain on the screen after the change. However, since Android wants to destroy and recreate the activity, that would have dire impacts on your dialog.

Pre-fragments, Android had a "managed dialog" facility that would attempt to help with this. However, with the introduction of fragments came the `DialogFragment`, which handles the configuration change process.

You have two ways of supplying the dialog to the `DialogFragment`:

1. You can override `onCreateDialog()` and return a `Dialog`, such as `AlertDialog` created via an `AlertDialog.Builder`
2. You can override `onCreateView()`, as you would with an ordinary fragment, and the `View` that you return will be placed inside of a dialog

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Dialogs/DialogFragment sample project demonstrates the use of a `DialogFragment` in conjunction with an `AlertDialog` in this fashion.

Here is our `DialogFragment`, named `SampleDialogFragment`:

```
package com.commonsware.android.dlgfrag;

import android.app.AlertDialog;
import android.app.Dialog;
import android.app.DialogFragment;
import android.content.DialogInterface;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

public class SampleDialogFragment extends DialogFragment implements
    DialogInterface.OnClickListener {
  private View form=null;

  @Override
  public Dialog onCreateDialog(Bundle savedInstanceState) {
    form=
        getActivity().getLayoutInflater()
                     .inflate(R.layout.dialog, null);

    AlertDialog.Builder builder=new AlertDialog.Builder(getActivity());

    return(builder.setTitle(R.string.dlg_title).setView(form)
                  .setPositiveButton(android.R.string.ok, this)
                  .setNegativeButton(android.R.string.cancel, null).create());
  }

  @Override
  public void onClick(DialogInterface dialog, int which) {
    String template=getActivity().getString(R.string.toast);
    EditText name=(EditText)form.findViewById(R.id.title);
    EditText value=(EditText)form.findViewById(R.id.value);
    String msg=
        String.format(template, name.getText().toString(),
                      value.getText().toString());

    Toast.makeText(getActivity(), msg, Toast.LENGTH_LONG).show();
  }

  @Override
  public void onDismiss(DialogInterface unused) {
    super.onDismiss(unused);

    Log.d(getClass().getSimpleName(), "Goodbye!");
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void onCancel(DialogInterface unused) {
    super.onCancel(unused);

    Toast.makeText(getActivity(), R.string.back, Toast.LENGTH_LONG).show();
  }
}
```

In `onCreateDialog()`, we inflate a custom layout (`R.layout.dialog`) that consists of some `TextView` labels and `EditText` fields:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    android:orientation="horizontal">

    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/display_name"/>

    <EditText
      android:id="@+id/title"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="text"/>
  </LinearLayout>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    android:orientation="horizontal">

    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/value"/>

    <EditText
      android:id="@+id/value"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="number"/>
  </LinearLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
</LinearLayout>
```

We then create an instance of `AlertDialog.Builder`, then start configuring the dialog by calling a series of methods on the `Builder`:

- `setTitle()` to supply the text to appear in the title bar of the dialog
- `setView()` to define the contents of the dialog, in the form of our inflated `View`
- `setPositiveButton()` to define the caption of one button (set here to the Android-supplied "OK" string resource) and to arrange to get control when that button is clicked (via `this` as the second parameter and our activity implementing `DialogInterface.OnClickListener`)
- `setNegativeButton()` to define the caption of the other button (set here to the Android-supplied "Cancel" resource)

We do not supply a listener to `setNegativeButton()`, because we do not need one in this case. Whenever the user clicks on *any* of the buttons, the dialog will be dismissed automatically. Hence, you only need a listener if you intend to do something special beyond dismissing the dialog when a button is clicked.

At that point, we call `create()` to construct the actual `AlertDialog` instance and hand that back to Android.

If the user taps our positive button, we are called with `onClick()` and can collect information from our form and do something with it, in this case displaying a `Toast`.

We also override:

- `onCancel()`, which is called if the user presses the BACK button to exit the dialog
- `onDismiss()`, which is called whenever the dialog goes away for any reason (BACK or a button click)

Our activity (`MainActivity`), has a big button tied to a `showMe()` method, which calls `show()` on a newly-created instance of our `SampleDialogFragment`:

```
  public void showMe(View v) {
    new SampleDialogFragment().show(getFragmentManager(), "sample");
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The second parameter to `show()` is a tag that can be used to retrieve this fragment again later from the `FragmentManager` via `findFragmentByTag()`.

When you click the big button in the activity, our dialog is displayed:

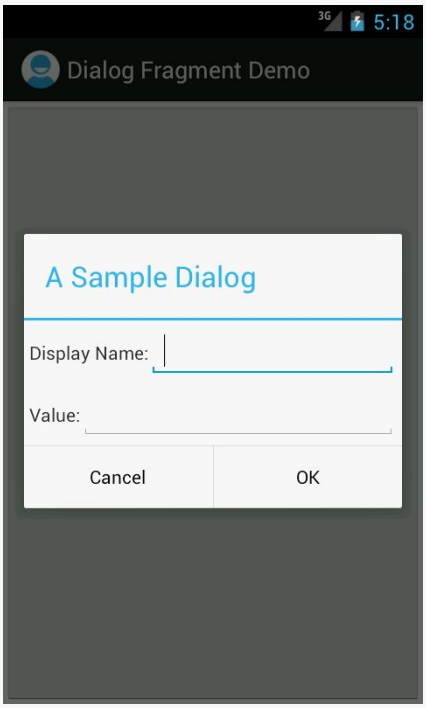


Figure 372: SampleDialogFragment, As Initially Launched, on Android 4.0.3

Android will handle the configuration change, and so long as our dialog uses typical widgets like `EditText`, the standard configuration change logic will carry our data forward from the old activity's dialog to the new activity's dialog.

DialogFragment: The Other Flavor

If you do not override `onCreateDialog()`, Android will assume that you want the `View` returned by `onCreateView()` to be poured into an ordinary `Dialog`, which `DialogFragment` will create for you automatically.

One advantage of this approach is that you can selectively show the fragment as a dialog *or* show it as a regular fragment as part of your main UI.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To show the fragment as a dialog, use the same `show()` technique as was outlined in the previous section. To display the fragment as part of the main UI, use a `FragmentTransaction` to `add()` it, the way you would for any other dynamic fragment.

This is one alternative to the normal fragment approach of having dedicated activities for each fragment on smaller screen sizes.

We will also see this approach used when we try to apply fragments to display content on a secondary screen using Android 4.2's `Presentation` class, covered elsewhere in this book.

Dialogs: Modal, Not Blocking

Dialogs in Android are modal in terms of UI. The user cannot proceed in your activity until they complete or dismiss the dialog.

Dialogs in Android are not blocking in terms of the programming model. When you call `show()` to display a dialog — either directly or by means of adding a `DialogFragment` to the screen — this is not a blocking call. The dialog will be displayed sometime after the call to `show()`, asynchronously. You use callbacks, such as the button event listeners, to find out about events going on with respect to the dialog that you care about.

This runs counter to a couple of GUI toolkits, where displaying the dialog blocks the thread that does the displaying. In those toolkits, the call to `show()` would not return until the dialog had been displayed and dealt with by the user. That being said, most modern GUI toolkits take the approach Android does and have dialogs be non-blocking. Some developers try to figure out some way of hacking a blocking approach on top of Android's non-blocking dialogs — their time would be far better spent learning modern event-driven programming.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Advanced ListViews

The humble `ListView` is the backbone of many an Android application. On phone-sized screens, the screen may be dominated by a single `ListView`, to allow the user to choose something to examine in more detail (e.g., pick a contact). On larger screens, the `ListView` may be shown side-by-side with the details of the selected item, to minimize the "pogo stick" effect seen on phones as users bounce back and forth between the list and the details.

While we have covered the basics of `ListView` in the core chapters of this book, there is a lot more that you can do if you so choose, to make your lists that much more interesting — this chapter will cover some of these techniques.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on `Adapter` and `AdapterView`.

Multiple Row Types, and Self Inflation

When we originally looked at `ListView`, we had all of our rows come from a common layout. Hence, while the data in each row would vary, the row structure itself would be consistent for all rows. This is very easy to set up, but it is not always what you want. Sometimes, you want a mix of row structures, such as header rows versus detail rows, or detail rows that vary a bit in structure based on the data:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

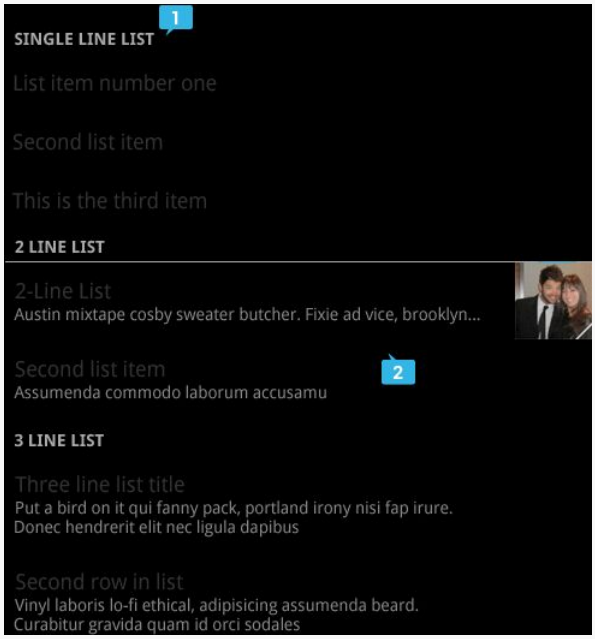


Figure 373: ListView with Row Structure Mix (image courtesy of Google)

Here, we see some header rows (e.g., "SINGLE LINE LIST") along with detail rows. While the detail rows visually vary a bit, they might still be all inflated from the same layout, simply making some pieces (second line of text, thumbnail, etc.) visible or invisible as needed. However, the header rows are sufficiently visually distinct that they really ought to come from separate layouts.

The good news is that Android supports multiple row types. However, this comes at a cost: you will need to handle the row creation yourself, rather than chaining to the superclass.

Our sample project, `Selection/HeaderDetailList` will demonstrate this, along with showing how you can create your own custom adapter straight from `BaseAdapter`, for data models that do not quite line up with what Android supports natively.

Our Data Model and Planned UI

The `HeaderDetailList` project is based on the `ViewHolderDemo` project from the chapter on `ListView`. However, this time, we have our list of 25 Latin words broken down into five groups of five, as seen in the `HeaderDetailList` activity:

```
  private static final String[][] items= {
      { "lorem", "ipsum", "dolor", "sit", "amet" },
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      { "consectetuer", "adipiscing", "elit", "morbi", "vel" },
      { "ligula", "vitae", "arcu", "aliquet", "mollis" },
      { "etiam", "vel", "erat", "placerat", "ante" },
      { "porttitor", "sodales", "pellentesque", "augue", "purus" } };
```

We want to display a header row for each batch:

Figure 374: HeaderDetailList, on Android 4.0.3

The Basic BaseAdapter

Once again, we have a custom `ListAdapter` named `IconicAdapter`. However, this time, instead of inheriting from `ArrayAdapter`, or even `CursorAdapter`, we are inheriting from `BaseAdapter`. As the name suggests, `BaseAdapter` is a basic implementation of the `ListAdapter` interface, with stock implementations of many of the `ListAdapter` methods. However, `BaseAdapter` is `abstract`, and so there are a few methods that we need to implement:

- `getCount()` returns the total number of rows that would be in the list. In our case, we total up the sizes of each of the batches, plus add one for each batch for our header rows:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public int getCount() {
    int count=0;

    for (String[] batch : items) {
      count+=1 + batch.length;
    }

    return(count);
  }
```

- `getItem()` needs to return the data model for a given position, passed in as the typical `int` index. An `ArrayAdapter` would return the value out of the array at that index; a `CursorAdapter` would return the `Cursor` positioned at that row. In our case, we will return one of two objects: either the `String` for rows that are to display a Latin word, or an `Integer` containing our batch's index for rows that are to be a header:

```
  @Override
  public Object getItem(int position) {
    int offset=position;
    int batchIndex=0;

    for (String[] batch : items) {
      if (offset == 0) {
        return(Integer.valueOf(batchIndex));
      }

      offset--;

      if (offset < batch.length) {
        return(batch[offset]);
      }

      offset-=batch.length;
      batchIndex++;
    }

    throw new IllegalArgumentException("Invalid position: "
        + String.valueOf(position));
  }
```

- `getItemId()` needs to return a unique `long` value for a given position. A `CursorAdapter` would find the `_id` value in the `Cursor` for that position and return it. In our case, lacking anything else, we simply return the position itself:

```
  @Override
  public long getItemId(int position) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(position);
  }
```

- `getView()`, which returns the `View` to use for a given row. This is the method that we overrode on our `IconicAdapter` in some previous incarnations to tailor the way the rows were populated. Our `getView()` implementation will be a bit more complex in this case, due to our multiple-row-type requirement, so we will examine it a bit later in this section.

Requesting Multiple Row Types

The methods listed above are the `abstract` ones that you have no choice but to implement yourself. Anything else on the `ListAdapter` interface that you wish to override you can, to replace the stub implementation supplied by `BaseAdapter`.

If you wish to have more than one type of row, there are two such methods that you will wish to override:

- `getViewTypeCount()` needs to return the number of distinct row types you will use. In our case, there are just two:

```
  @Override
  public int getViewTypeCount() {
    return(2);
  }
```

- `getItemViewType()` needs to return a value from `0` to `getViewTypeCount()-1`, indicating the index of the particular row type to use for a particular row position. In our case, we need to return different values for headers (`0`) and detail rows (`1`). To determine which is which, we use `getItem()` — if we get an `Integer` back, we need to use a header row for that position:

```
  @Override
  public int getItemViewType(int position) {
    if (getItem(position) instanceof Integer) {
      return(0);
    }

    return(1);
  }
```

The reason for supplying this information is for row recycling. The `View` that is passed into `getView()` is either `null` or a row that we had previously created that has scrolled off the screen. By passing us this now-unused `View`, Android is asking us to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

reuse it if possible. By specifying the row type for each position, Android will ensure that it hands us the right type of row for recycling — we will not be passed in a header row to recycle when we need to be returning a detail row, for example.

Creating and Recycling the Rows

Our `getView()` implementation, then, needs to have two key enhancements over previous versions:

1. We need to create the rows ourselves, particularly using the appropriate layout for the required row type (header or detail)
2. We need to recycle the rows when they are provided, as this has a **major** impact on the scrolling speed of our `ListView`

To help simplify the logic, we will have `getView()` focus on the detail rows, with a separate `getHeaderView()` to create/recycle and populate the header rows. Our `getView()` determines up front whether the row required is a header and, if so, delegates the work to `getHeaderView()`:

```
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      if (getItemViewType(position) == 0) {
        return(getHeaderView(position, convertView, parent));
      }

      View row=convertView;

      if (row == null) {
        row=getLayoutInflater().inflate(R.layout.row, parent, false);
      }

      ViewHolder holder=(ViewHolder)row.getTag();

      if (holder == null) {
        holder=new ViewHolder(row);
        row.setTag(holder);
      }

      String word=(String)getItem(position);

      if (word.length() > 4) {
        holder.icon.setImageResource(R.drawable.delete);
      }
      else {
        holder.icon.setImageResource(R.drawable.ok);
      }

      holder.label.setText(word);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      holder.size.setText(String.format(getString(R.string.size_template),
                                        word.length()));

      return(row);
    }
```

Assuming that we are to create a detail row, we then check to see if we were passed in a non-`null` `View`. If we were passed in `null`, we cannot recycle that row, so we have to inflate a new one via a call to `inflate()` on a `LayoutInflater` we get via `getLayoutInflater()`. But, if we were passed in an actual `View` to recycle, we can skip this step.

From here, the `getView()` implementation is largely the way it was before, including dealing with the `ViewHolder`. The only change of significance is that we have to manage the `label` `TextView` ourselves — before, we chained to the superclass and let `ArrayAdapter` handle that. So our `ViewHolder` now has a `label` data member with our `label` `TextView`, and we fill it in along with the `size` and `icon`. Also, we use `getItem()` to retrieve our Latin word, so it can find the right word for the given position out of our various word batches.

Our `getHeaderView()` does much the same thing, except it uses `getItem()` to retrieve our batch index, and we use that for constructing our header:

```
    private View getHeaderView(int position, View convertView,
                               ViewGroup parent) {
      View row=convertView;

      if (row == null) {
        row=getLayoutInflater().inflate(R.layout.header, parent, false);
      }

      Integer batchIndex=(Integer)getItem(position);
      TextView label=(TextView)row.findViewById(R.id.label);

      label.setText(String.format(getString(R.string.batch),
                                  1 + batchIndex.intValue()));

      return(row);
    }
```

Choice Modes and the Activated Style

In the chapter on large-screen strategies, we saw the EU4You sample application, and we mentioned that the `ListView` formatted its rows as "activated" to represent the current selection, when the `ListView` was side-by-side with the details.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In the chapter on styles, we saw an example of an "activated" style that referred to a device-specific color to use for an activated background. It just so happens that this is the same style that we used in EU4You.

Hence, the recipe for using activated notation for a `ListView` adjacent to details on the last-clicked-upon `ListView` row is:

- Use `CHOICE_MODE_SINGLE` (or `android:choiceMode="singleChoice"`) on the `ListView`.
- Have a style resource, in `res/values-v11/`, that references the device-specific activated background:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated" parent="android:Theme.Holo">
    <item name="android:background">?android:attr/
activatedBackgroundIndicator</item>
  </style>
</resources>
```

- Have the same style resource also defined in `res/values` if you are supporting pre-Honeycomb devices, where you skip the parent and the background color override, as neither of those specific values existed before API Level 11:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated">
  </style>
</resources>
```

- Use that style as the background of your `ListView` row (e.g., `style="@style/activated"`)

Android will automatically color the row background based upon the last row clicked, instead of checking a `RadioButton` as you might ordinarily see with `CHOICE_MODE_SINGLE` lists.

Custom Mutable Row Contents

Lists with pretty icons next to them are all fine and well. But, can we create `ListView` widgets whose rows contain interactive child widgets instead of just passive widgets like `TextView` and `ImageView`? For example, there is a `RatingBar` widget that allows users to assign a rating by clicking on a set of star icons. Could we combine the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

RatingBar with text in order to allow people to scroll a list of, say, songs and rate them right inside the list?

There is good news and bad news.

The good news is that interactive widgets in rows work just fine. The bad news is that it is a little tricky, specifically when it comes to taking action when the interactive widget's state changes (e.g., a value is typed into a field). We need to store that state somewhere, since our RatingBar widget will be recycled when the ListView is scrolled. We need to be able to set the RatingBar state based upon the actual word we are viewing as the RatingBar is recycled, and we need to save the state when it changes so it can be restored when this particular row is scrolled back into view.

What makes this interesting is that, by default, the RatingBar has absolutely no idea what item in the ArrayAdapter it represents. After all, the RatingBar is just a widget, used in a row of a ListView. We need to teach the rows which item in the ArrayAdapter they are currently displaying, so when their RatingBar is checked, they know which item's state to modify.

So, let's see how this is done, using the activity in the Selection/RateList sample project. We will use the same basic classes as in most of our ListView samples, where we are showing a list of Latin words. In this case, you can rate the words on a three-star rating. Words given a top rating are put in all caps:

```
package com.commonsware.android.ratelist;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.LinearLayout;
import android.widget.RatingBar;
import android.widget.TextView;
import java.util.ArrayList;

public class RateListDemo extends ListActivity {
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  ArrayList<RowModel> list=new ArrayList<RowModel>();

  for (String s : items) {
    list.add(new RowModel(s));
  }

  setListAdapter(new RatingAdapter(list));
}

private RowModel getModel(int position) {
  return(((RatingAdapter)getListAdapter()).getItem(position));
}

class RatingAdapter extends ArrayAdapter<RowModel> {
  RatingAdapter(ArrayList<RowModel> list) {
    super(RateListDemo.this, R.layout.row, R.id.label, list);
  }

  public View getView(int position, View convertView,
                      ViewGroup parent) {
    View row=super.getView(position, convertView, parent);
    RatingBar bar=(RatingBar)row.getTag();

    if (bar==null) {
      bar=(RatingBar)row.findViewById(R.id.rate);
      row.setTag(bar);

      RatingBar.OnRatingBarChangeListener l=
                  new RatingBar.OnRatingBarChangeListener() {
        public void onRatingChanged(RatingBar ratingBar,
                                    float rating,
                                    boolean fromTouch)  {
          Integer myPosition=(Integer)ratingBar.getTag();
          RowModel model=getModel(myPosition);

          model.rating=rating;

          LinearLayout parent=(LinearLayout)ratingBar.getParent();
          TextView label=(TextView)parent.findViewById(R.id.label);

          label.setText(model.toString());
        }
      };

      bar.setOnRatingBarChangeListener(l);
    }

    RowModel model=getModel(position);

    bar.setTag(Integer.valueOf(position));
    bar.setRating(model.rating);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      return(row);
    }
  }

  class RowModel {
    String label;
    float rating=2.0f;

    RowModel(String label) {
      this.label=label;
    }

    public String toString() {
      if (rating>=3.0) {
        return(label.toUpperCase());
      }

      return(label);
    }
  }
}
```

Here is what is different in this activity and `getView()` implementation than in earlier, simpler samples:

1. While we are still using `String` array items as the list of Latin words, rather than pour that String array straight into an `ArrayAdapter`, we turn it into a list of `RowModel` objects. `RowModel` is the mutable model: it holds the Latin word plus the current rating. In a real system, these might be objects populated from a database, and the properties would have more business meaning.
2. Utility methods like `onListItemClick()` had to be updated to reflect the change from a pure-String model to use a `RowModel`.
3. The `ArrayAdapter` subclass (`RatingAdapter`), in `getView()`, lets `ArrayAdapter` inflate and recycle the row, then checks to see if we have a `ViewHolder` in the row's tag. If not, we create a new `ViewHolder` and associate it with the row. For the row's `RatingBar`, we add an anonymous `onRatingChanged()` listener that looks at the row's tag (`getTag()`) and converts that into an `Integer`, representing the position within the `ArrayAdapter` that this row is displaying. Using that, the rating bar can get the actual `RowModel` for the row and update the model based upon the new state of the rating bar. It also updates the text adjacent to the `RatingBar` when checked to match the rating bar state.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

4. We always make sure that the `RatingBar` has the proper contents and has a tag (via `setTag()`) pointing to the position in the adapter the row is displaying.

The row layout is very simple: just a `RatingBar` and a `TextView` inside a `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal"
>
  <RatingBar
    android:id="@+id/rate"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:numStars="3"
    android:stepSize="1"
    android:rating="2" />
  <TextView
    android:id="@+id/label"
    android:padding="2dip"
    android:textSize="18sp"
    android:layout_gravity="left|center_vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
</LinearLayout>
```

And the result is what you would expect, visually:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 375: RateList, As Initially Shown

This includes the toggled rating bars turning their words into all caps:

Figure 376: RateList, With a Three-Star Word

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

From Head To Toe

Perhaps you do not need section headers scattered throughout your list. If you only need extra "fake rows" at the beginning or end of your list, you can use header and footer views.

`ListView` supports `addHeaderView()` and `addFooterView()` methods that allow you to add `View` objects to the beginning and end of the list, respectively. These `View` objects otherwise behave like regular rows, in that they are part of the scrolled area and will scroll off the screen if the list is long enough. If you want fixed headers or footers, rather than put them in the `ListView` itself, put them outside the `ListView`, perhaps using a `LinearLayout`.

To demonstrate header and footer views, take a peek at the `Selection/HeaderFooter` sample project, particularly the `HeaderFooterDemo` class:

```
package com.commonsware.android.header;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import android.app.ListActivity;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.TextView;

public class HeaderFooterDemo extends ListActivity {
  private static String[] items={"lorem", "ipsum", "dolor",
                                 "sit", "amet", "consectetuer",
                                 "adipiscing", "elit", "morbi",
                                 "vel", "ligula", "vitae",
                                 "arcu", "aliquet", "mollis",
                                 "etiam", "vel", "erat",
                                 "placerat", "ante",
                                 "porttitor", "sodales",
                                 "pellentesque", "augue",
                                 "purus"};
  private long startTime=SystemClock.uptimeMillis();
  private boolean areWeDeadYet=false;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    getListView().addHeaderView(buildHeader());
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    getListView().addFooterView(buildFooter());
    setListAdapter(new ArrayAdapter<String>(this,
                        android.R.layout.simple_list_item_1,
                        items));
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    areWeDeadYet=true;
  }

  private View buildHeader() {
    Button btn=new Button(this);

    btn.setText("Randomize!");
    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        List<String> list=Arrays.asList(items);

        Collections.shuffle(list);

        setListAdapter(new ArrayAdapter<String>(HeaderFooterDemo.this,
                            android.R.layout.simple_list_item_1,
                            list));
      }
    });

    return(btn);
  }

  private View buildFooter() {
    TextView txt=new TextView(this);

    updateFooter(txt);

    return(txt);
  }

  private void updateFooter(final TextView txt) {
    long runtime=(SystemClock.uptimeMillis()-startTime)/1000;

    txt.setText(String.valueOf(runtime)+" seconds since activity launched");

    if (!areWeDeadYet) {
      getListView().postDelayed(new Runnable() {
        public void run() {
          updateFooter(txt);
        }
      }, 1000);
    }
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here, we add a header `View` built via `buildHeader()`, returning a `Button` that, when clicked, will shuffle the contents of the list. We also add a footer `View` built via `buildFooter()`, returning a `TextView` that shows how long the activity has been running, updated every second. The list itself is the ever-popular list of *lorem ipsum* words.

When initially displayed, the header is visible but the footer is not, because the list is too long:

Figure 377: A ListView with a header view shown

If you scroll downward, the header will slide off the top, and eventually the footer will scroll into view:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 378: A ListView with a footer view shown

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Action Bar Navigation

Beyond the app icon (a.k.a., icon on the left), action bar toolbar items, and the overflow menu, the action bar also supports a navigation area. This resides to the right of the app icon and to the left of the toolbar items/overflow menu. You can:

- Put tabs in here, to allow users to switch between portions of your app
- Use "list navigation", which effectively puts a `Spinner` in here, also to allow users to switch from place to place
- Put in some other custom form of navigation, such as a search field

This chapter will review how to do these things, and how they tie into other constructs in Android, notably the `ViewPager`.

NOTE: The tab and list navigation forms outlined in this chapter have been marked as deprecated in Android 5.0. Developers are encouraged to use other techniques for in-activity navigation.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on the action bar.

List Navigation

Android's action bar supports a "list navigation" option. Despite the name, the "list" is really a `Spinner`, hosted in the action bar. You get to populate the `Spinner` via your own `SpinnerAdapter`, and you get control when the user changes the selected item, so that you can update your UI as you see fit.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To set this up:

1. Call `setNavigationMode(ActionBar.NAVIGATION_MODE_LIST)` on the `ActionBar` to enable the list navigation mode, which you get via `getActionBar()`
2. Call `setListNavigationCallbacks()` on the `ActionBar`, simultaneously supplying the `SpinnerAdapter` to use to populate the `Spinner` and an `ActionBar.OnNavigationListener` object to be notified when there is a selection change in the `Spinner`

The `ActionBar/ListNavNative` sample project demonstrates this, using a variation on the "whole lot of editors" UI first seen in the `ViewPager` chapter.

We want to display a full-screen `EditText` widget whose contents will be driven by the list navigation selection. The fragment for this — `EditorFragment` — is a slightly revised version of the same class from the `ViewPager` samples. Here, though, state management will be handled completely by the activity, so we simply expose getters and setters as needed for working with the text in the editor, along with its hint:

```
package com.commonsware.android.listnav;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;

public class EditorFragment extends Fragment {
  private EditText editor=null;

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.editor, container, false);

    editor=(EditText)result.findViewById(R.id.editor);

    return(result);
  }

  CharSequence getText() {
    return(editor.getText());
  }

  void setText(CharSequence text) {
    editor.setText(text);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }

  void setHint(CharSequence hint) {
    editor.setHint(hint);
  }
}
```

Setting up the list navigation mode is part of the work we do in `onCreate()`:

```
    ArrayAdapter<String> nav=null;
    ActionBar bar=getActionBar();

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH) {
      nav=
          new ArrayAdapter<String>(
                                    bar.getThemedContext(),
                                    android.R.layout.simple_spinner_item,
                                    labels);
    }
    else {
      nav=
          new ArrayAdapter<String>(
                                    this,
                                    android.R.layout.simple_spinner_item,
                                    labels);
    }

    nav.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
    bar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
    bar.setListNavigationCallbacks(nav, this);
```

Android 4.0 added a `getThemedContext()` method on `ActionBar`. Use the `Context` returned by this method when working with resources that relate to the `ActionBar`. In this case, we use it when creating our `ArrayAdapter` to use with the `Spinner`. However, since this is only available on API Level 14 and higher, you need to check for that and fall back to using the `Activity` as your `Context` for earlier versions of Android.

We then use `setNavigationMode()` to indicate that we want list navigation, then use `setListNavigationCallbacks()` to supply our `ArrayAdapter`, plus our implementation of `OnNavigationListener` — in this case, we are implementing this interface on the activity itself.

Because we are implementing `OnNavigationListener`, we need to override the `onNavigationItemSelected()` method. This will get called when the `Spinner` selection changes (including when it is initially set), and it is up to us to affect our UI. That requires a bit of additional preparation work:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- We set up our `EditorFragment` in `onCreate()`, if it does not already exist:

```
    frag=
(EditorFragment)getFragmentManager().findFragmentById(android.R.id.content);

    if (frag==null) {
      frag=new EditorFragment();
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content, frag)
                          .commit();
    }
```

- We track the last known position of the `Spinner` selection, by means of a `lastPosition` data member
- We store our data model (the text held by the editor) in a `models` `CharSequence` array

Our objective is to have 10 total "editors", accessible via the list navigation. Our `labels` array in our `ArrayAdapter` has 10 entries, and `models` is a 10-item array to match.

That allows us to implement `onNavigationItemSelected()`:

```
  @Override
  public boolean onNavigationItemSelected(int itemPosition, long itemId) {
    if (lastPosition > -1) {
      models[lastPosition]=frag.getText();
    }

    lastPosition=itemPosition;
    frag.setText(models[itemPosition]);
    frag.setHint(labels[itemPosition]);

    return(true);
  }
```

In the `ViewPager` sample, we actually had 10 instances of `EditorFragment`. Here, we have just one. We will use it for all 10 positions. Hence, all we do is grab the current contents of the editor and save them in `models` (except when we are first starting and have no prior position). Then, we populate the editor with the next model and a suitable hint.

Now, we *could* have 10 instances of `EditorFragment` and swap between them with `FragmentTransactions`. Or, we could have a variety of distinct fragment instances, from different classes, and swap between them using `FragmentTransactions`. What you do to update your UI based upon the list navigation change is up to you.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

One limitation of list navigation, compared to `ViewPager`, is state management on configuration changes. `ViewPager` handled keeping track of what page we were on, and if we retained all our fragments, our model data (the editors' contents) were retained as well. With list navigation and a single non-retained fragment, we have to do all of that ourselves.

So, we implement `onSaveInstanceState()` to persist both the `models` array and our current position:

```
  @Override
  public void onSaveInstanceState(Bundle state) {
    if (lastPosition > -1) {
      models[lastPosition]=frag.getText();
    }

    state.putCharSequenceArray(KEY_MODELS, models);
    state.putInt(KEY_POSITION,
                 getActionBar().getSelectedNavigationIndex());
  }
```

In `onCreate()`, we restore our `models` array:

```
    if (state != null) {
      models=state.getCharSequenceArray(KEY_MODELS);
    }
```

And, later in `onCreate()`, we tell the action bar which position to select:

```
    if (state != null) {
      bar.setSelectedNavigationItem(state.getInt(KEY_POSITION));
    }
```

The result is a `Spinner` in the action bar, allowing the user to choose which of the 10 "editors" to work with:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 379: ListNavDemo, Showing the List, on Android 4.0.3

Tabs (And Sometimes List) Navigation

Similarly, you can set up tab navigation, where you present a roster of tabs the user can tap on.

Maybe.

(We'll get to the explanation of "maybe" in a bit)

Setting up tabs is fairly straightforward, once you know the recipe:

1. Call `setNavigationMode(ActionBar.NAVIGATION_MODE_TABS)` on the `ActionBar`, which you get via `getActionBar()`
2. Call `addTab()` on `ActionBar` for each tab you want, supplying at minimum the text caption of the tab and a `TabListener` implementation that will be notified of state changes in that tab

The `ActionBar/TabFragmentDemoNative` sample project is very similar to the one for list navigation described above, except that it uses tabs instead of list navigation. We

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

have the same 10 editors, the same data model (`models`), and the same basic logic for saving and restoring our instance state. What differs is in how we set up the UI.

As with list navigation, you can do whatever you want when tabs are selected or unselected. You could:

- Add and remove fragments
- Attach and detach fragments (which remove them from the UI but keep them in the `FragmentManager` for later reuse)
- Flip pages of a `ViewPager`
- Update a simple UI in place (akin to what we did in the list navigation sample above)

In our case, we will take the "caveman" approach of replacing our entire fragment on each tab click.

Our `EditorFragment` is a bit closer to the original from the `ViewPager` samples, except that this time we pass in the initial text to display, along with the position, in the factory method:

```
package com.commonsware.android.tabfrag;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;

public class EditorFragment extends Fragment {
  private static final String KEY_POSITION="position";
  private static final String KEY_TEXT="text";
  private EditText editor=null;

  static EditorFragment newInstance(int position,
                                    CharSequence text) {
    EditorFragment frag=new EditorFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_POSITION, position);
    args.putCharSequence(KEY_TEXT, text);
    frag.setArguments(args);

    return(frag);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                          ViewGroup container,
                          Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.editor, container, false);

    editor=(EditText)result.findViewById(R.id.editor);

    int position=getArguments().getInt(KEY_POSITION, -1);

    editor.setHint(String.format(getString(R.string.hint), position + 1));
    editor.setText(getArguments().getCharSequence(KEY_TEXT));

    return(result);
  }

  CharSequence getText() {
    return(editor.getText());
  }
}
```

In `onCreate()`, we tell the `ActionBar` that we want tab navigation, then we add 10 tabs to the bar:

```
    ActionBar bar=getActionBar();
    bar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

    for (int i=0; i < 10; i++) {
      bar.addTab(bar.newTab().setText("Tab #" + String.valueOf(i + 1))
                    .setTabListener(this).setTag(i));
    }
```

Calling `newTab()` on the `ActionBar` gives us an `ActionBar.Tab` object, which we can use builder-style to configure the tab. In our case, we are setting the caption (`setText()`), the listener (`setTabListener()`), and a tag to use to identify this tab (`setTag()`). The tag is akin to the tags on `Views` — it can be any object you want. In our case, we just use the index of the tab.

Our activity needs to implement the `TabListener` interface, since we are passing it into the `setTabListener()` method. There are three methods you must implement on that interface:

1. `onTabSelected()` is called when the tab is selected by the user
2. `onTabUnselected()` is called when some other tab is selected by the user
3. `onTabReselected()` is called, presumably, when the user taps on an already-selected tab (e.g., to refresh the tab's contents)

Our implementation ignores the latter and focuses on the first two:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void onTabSelected(Tab tab, FragmentTransaction ft) {
    int i=((Integer)tab.getTag()).intValue();

    ft.replace(android.R.id.content,
               EditorFragment.newInstance(i, models[i]));
  }

  @Override
  public void onTabUnselected(Tab tab, FragmentTransaction ft) {
    int i=((Integer)tab.getTag()).intValue();
    EditorFragment frag=

(EditorFragment)getFragmentManager().findFragmentById(android.R.id.content);

    if (frag != null) {
      models[i]=frag.getText();
    }
  }

  @Override
  public void onTabReselected(Tab tab, FragmentTransaction ft) {
    // unused
  }
```

In `onTabSelected()`, we get our tab's position via its tag, then call `replace()` on the supplied `FragmentTransaction` to replace the current contents of the activity with a new `EditorFragment`, set up with the proper position and model data.

In `onTabUnselected()`, we get our tab's position and the `EditorFragment`, then save the updated text (if any) from the editor in `models` for later reuse.

Running this on a phone-sized screen gives you your tabs, in a row beneath the main action bar itself:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

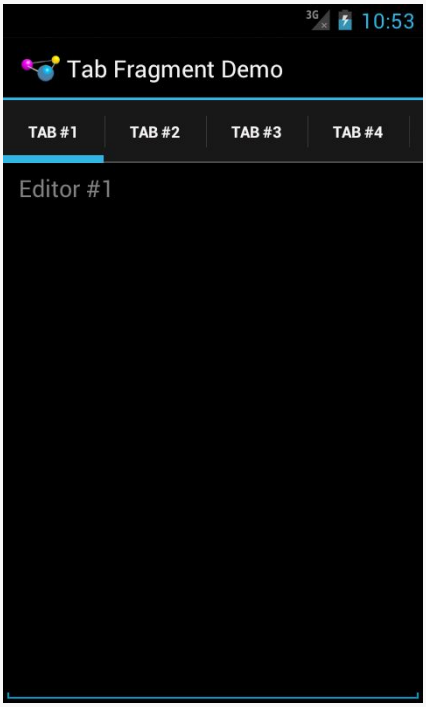


Figure 380: TabFragmentDemo, on Android 4.0.3, Phone-Sized Screen

Those tabs are "swipey", meaning that the user can fling the row of tabs to get to all 10 of them.

This UI makes perfect sense for something described as "tab navigation". Where things get a bit odd is in any configuration, such as a normal-sized screen in landscape:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

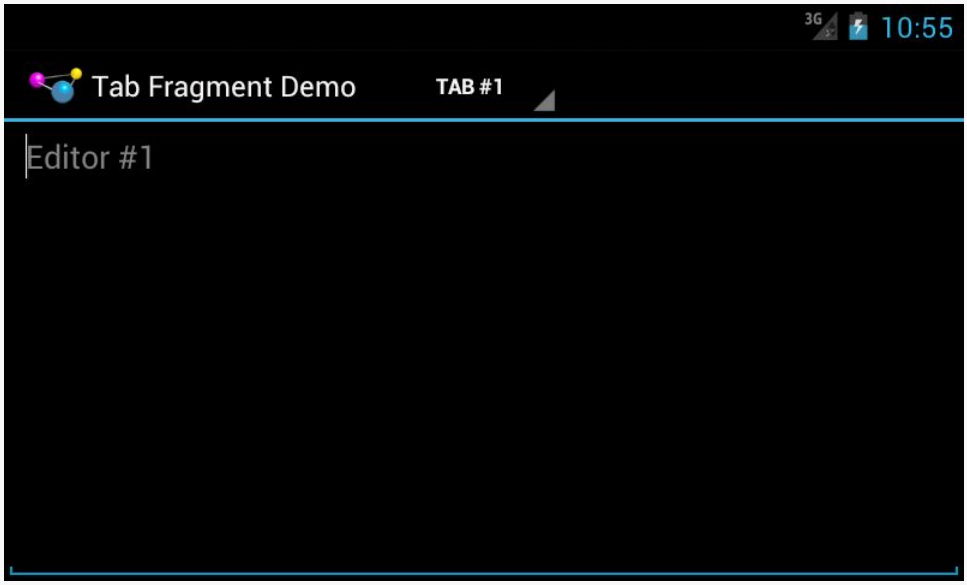


Figure 381: TabFragmentDemo, on Android 4.0.3, Phone-Sized Screen in Landscape

or on a large-sized screen in portrait:

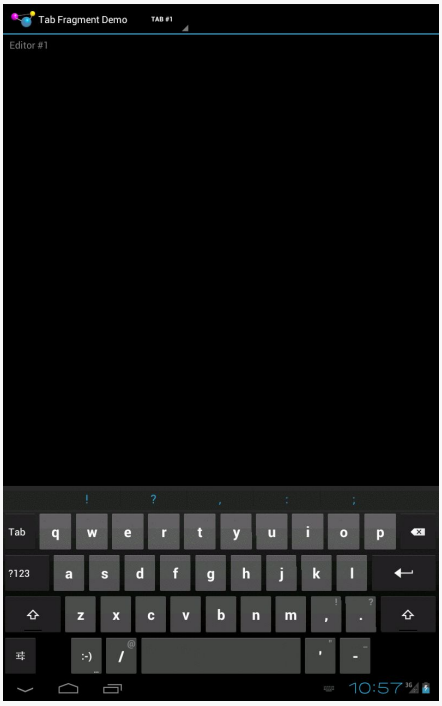


Figure 382: TabFragmentDemo, on Android 4.0.3, Tablet-Sized Screen in Portrait

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android will automatically convert your tab navigation to list navigation if and when it wishes to. You do not have control over this behavior, and it will vary by Android release:

> The system will apply the correct UX policy for the device. As the exact policy of presentation may change on different devices or in future releases, it is intentionally not specified in documentation.

(from the issue filed by the author of this book over this behavior)

Custom Navigation

You could also elect to use one of the various flavors of `setCustomView()` on `ActionBar`. These allow you to completely control what goes in the navigation area of the bar, by supplying either a `View` or a layout resource ID that should get inflated into the bar. Particularly in the latter case, you would call `getCustomView()` later on to retrieve the inflated layout, so you can access the widgets, configure listeners, and so forth.

While Google definitely steers you in the direction of using the tabs or list navigation, plenty of apps will use a custom navigation option, for things like:

- a search field
- an `AutoCompleteTextView` (e.g., a browser's address bar)
- etc.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Action Modes and Context Menus

If you have spent much time on an Android 3.0+ device, then you probably have run into a curious phenomenon. Sometimes, when you select an item in a list or other widget, the action bar magically transforms from its normal look:

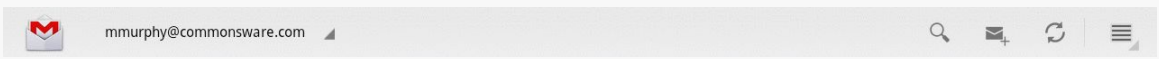


Figure 383: The Gmail action bar on a Honeycomb tablet, in normal mode

to one designed to perform operations on what you have selected:

Figure 384: The Gmail action bar on a Honeycomb tablet, showing an action mode

The good news is that this is not some sort of magic limited only to firmware applications like Gmail. You too can have this effect in your application, by triggering an "action mode".

Action modes — sometimes called the "contextual action bar" — is the replacement for the "context menu", whereby a menu would appear when you long-tap on some widget. Context menus were most commonly used with `AdapterViews`, particularly with `ListView`, to perform an operation on the specific long-tapped-upon item.

In this chapter, we will explore action modes, with context menus being relegated to the dustbin of history.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on the action bar.

Another Wee Spot O' History

Most desktop operating systems have had the notion of a "context menu" for some time, typically triggered by a click of the right mouse button. In particular, a right-click over some selected item might bring up a context menu of operations to perform on that item:

- Selecting text in a text editor, then right-clicking, might bring up a context menu for cut/copy/paste of the text
- Right-clicking over a file in some sort of file explorer might bring up a context menu for cut/copy/paste of the file
- Etc.

Android supports context menus, driven by a long-tap on a widget rather than a right-click. You will find many applications that offer such menus, particularly on lists of things.

Context menus are certainly useful. Power users can save screen taps if they know where context menus reside and what features they offer. For example, rather than tapping on a list item, then opening an options menu, then tapping a menu item to delete something, a power user could long-tap to open a context menu, then tap on a context menu item — saving one tap and switching back and forth between activities.

The problem is that context menus are invisible and are triggered by an action not used elsewhere very much (long tap).

In theory, users would find out about context menus in your application from reading your documentation. That would imply that we were in some alternate universe where all users read documentation, all people live in peace and harmony, and all book authors have great heads of hair. In this universe, power users will find your context menus, but ordinary users may be completely oblivious to them. Also, the hair of book authors remains stubbornly variable.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The action bar itself is designed to help raise the visibility of what had been the options menu (e.g., turning menu items into toolbar buttons) and standardizing the location of navigation elements (e.g., tabs, search fields). The action mode is designed to perform a similar bit of magic for context menus. Rather than have context menus be buried under a long-tap, action modes let the contextual actions take over the action bar, putting them front-and-center in the user experience.

Manual Action Modes

A common pattern will be to activate an action mode when the user checks off something in a multiple-choice `ListView`, as is the case with applications like Gmail. If you want to go that route, there is some built-in scaffolding to make that work, described later in this chapter.

You can, if you wish, move the action bar into an action mode whenever you want. This would be particularly important if your UI is not based on a `ListView`. For example, tapping on an image in a `GridView` might activate it and move you into an action mode for operations upon that particular image.

In this section, we will examine the `ActionMode/ManualNative` sample project. This is another variation on the "show a list of Latin words in a list" sample used elsewhere in this book.

Choosing Your Trigger

As noted above, Gmail switches into an action mode when the user checks off one or more conversations in the conversations list. Selecting a word or passage in an `EditText` (e.g., via a long-tap) brings up an action mode for cut/copy/paste operations. And so on.

You will need to choose, for your own UI, what trigger mechanism will bring up an action mode. It should be some trigger that makes it obvious to the user what the action mode will be acting upon. For example:

- If the user long-taps on an item in a `GridView`, bring up an action mode, and treat future taps on `GridView` items as adding or removing items from the "selection" while that action mode is visible
- If the user "rubber-bands" some figures in your vector art drawing `View`, bring up an action mode for operations on those figures (e.g., rotate, resize)
- And so on

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In the case of the sample project, we stick with the classic long-tap on a `ListView` row to bring up an action mode that replaces the context menu when run on a API Level 11+ device:

```
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    initAdapter();
    getListView().setLongClickable(true);
    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
    getListView().setOnItemLongClickListener(new ActionModeHelper(
                                                                  this,

getListView()));
  }
```

Starting the Action Mode

Starting an action mode is trivially easy: just call `startActionMode()` on your `Activity`, passing in an implementation of `ActionMode.Callback`, which will be called with various lifecycle methods for the action mode itself.

In the case of the `ActionMode` sample project, `ActionModeHelper` – our `OnItemLongClickListener` from the preceding section – also is our `ActionMode.Callback` implementation. Hence, when the user long-clicks on an item in the `ListView`, the `ActionModeHelper` establishes itself as the action mode:

```
  @Override
  public boolean onItemLongClick(AdapterView<?> view, View row,
                                 int position, long id) {
    modeView.clearChoices();
    modeView.setItemChecked(position, true);

    if (activeMode == null) {
      activeMode=host.startActionMode(this);
    }

    return(true);
  }
```

Note that `startActionMode()` returns an `ActionMode` object, which we can use later on to configure the mode's behavior, by stashing it in an `actionMode` data member.

Also, we make the long-clicked-upon item be "checked", to show which item the action mode will act upon. Our row layout will make a checked row show up with

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the "activated" style, courtesy of Android's `simple_list_item_activated_1` stock layout.

Also note that we only start the action mode if it is not already started.

Implementing the Action Mode

The real logic behind the action mode lies in your `ActionMode.Callback` implementation. It is in these four lifecycle methods where you define what the action mode should look like and what should happen when choices are made in it.

onCreateActionMode()

The `onCreateActionMode()` method will be called shortly after you call `startActionMode()`. Here, you get to define what goes in the action mode. You get the `ActionMode` object itself (in case you do not already have a reference to it). More importantly, you are passed a `Menu` object, just as you get in `onCreateOptionsMenu()`. And, just like with `onCreateOptionsMenu()`, you can inflate a menu resource into the `Menu` object to define the contents of the action mode:

```
@Override
public boolean onCreateActionMode(ActionMode mode, Menu menu) {
  MenuInflater inflater=host.getMenuInflater();

  inflater.inflate(R.menu.context, menu);
  mode.setTitle(R.string.context_title);

  return(true);
}
```

In addition to inflating our context menu resource into the action mode's menu, we also set the title of the `ActionMode`, which shows up to the right of the Done button:

Figure 385: The ActionMode sample application's action bar on a Honeycomb tablet, showing the active action mode

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

onPrepareActionMode()

If you determine that you need to change the contents of your action mode, you can call `invalidate()` on the `ActionMode` object. That, in turn, will trigger a call to `onPrepareActionMode()`, where you once again have an opportunity to configure the `Menu` object. If you do make changes, return `true` — otherwise, return `false`. In the case of `ActionModeHelper`, we take the latter approach:

```
@Override
public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
  return(false);
}
```

onActionItemClicked()

Just as `onCreateActionMode()` is the action mode analogue to `onCreateOptionsMenu()`, `onActionItemClicked()` is the action mode analogue to `onOptionsItemSelected()`. This will be called if the user clicks on something related to your action mode. You are passed in the corresponding `MenuItem` object (plus the `ActionMode` itself), and you can take whatever steps are necessary to do whatever the work is.

On the `ActionModeDemo` class, we have the business logic for handling the data-change operations in a `performAction()` method:

```
public boolean performAction(int itemId, int position) {
  switch (itemId) {
    case R.id.cap:
      String word=words.get(position);

      word=word.toUpperCase();

      adapter.remove(words.get(position));
      adapter.insert(word, position);

      return(true);

    case R.id.remove:
      adapter.remove(words.get(position));

      return(true);
  }

  return(false);
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

And, the `onActionItemClicked()` method calls `performAction()`:

```
@Override
public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
  boolean result=
      host.performAction(item.getItemId(),
                         modeView.getCheckedItemPosition());

  if (item.getItemId() == R.id.remove) {
    activeMode.finish();
  }

  return(result);
}
```

`onActionItemClicked()` also dismisses the action mode if the user chose the "remove" item, since the action mode is no longer needed. You get rid of an active action mode by calling `finish()` on it.

onDestroyActionMode()

The `onDestroyActionMode()` callback will be invoked when the action mode goes away, for any reason, such as:

1. The user clicks the Done button on the left
2. The user clicks the BACK button
3. You call `finish()` on the `ActionMode`

Here, you can do any necessary cleanup. `ActionModeHelper` tries to clean things up, notably the "checked" state of the last item long-tapped-upon:

```
@Override
public void onDestroyActionMode(ActionMode mode) {
  activeMode=null;
  modeView.clearChoices();
  modeView.requestLayout();
}
```

However, for reasons that are not yet clear, `clearChoices()` does not update the UI when called from `onDestroyActionMode()` unless you also call `requestLayout()`.

Multiple-Choice-Modal Action Modes

For many cases, the best user experience will be for you to have a multiple-choice `ListView`, where checking items in that list enables an action mode for performing

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

operations on the checked items. For this scenario, API Level 11+ has a built-in `ListView` choice mode, `CHOICE_MODE_MULTIPLE_MODAL`, that automatically sets up an `ActionMode` for you as the user checks and unchecks items.

To see how this works, let's examine the ActionMode/ActionModeMC sample project. This is the same project as in the preceding section, but altered to have a multiple-choice `ListView`, utilizing an action mode on API Level 11+.

Once again, in `onCreate()`, we need to set up the smarts for our `ListView`. This time, though, we will use `CHOICE_MODE_MULTIPLE_MODAL`:

```
@Override
 public void onCreate(Bundle icicle) {
   super.onCreate(icicle);

   initAdapter();

   getListView().setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
   getListView().setMultiChoiceModeListener(new HCMultiChoiceModeListener(
       this, getListView()));
 }
```

We enable `CHOICE_MODE_MULTIPLE_MODAL` for the `ListView`, and register an instance of an `HCMultiChoiceModeListener` object via `setMultiChoiceModeListener()`. This object is an implementation of the `MultiChoiceModeListener` interface that we will examine shortly.

Since we now may have multiple checked items, our `performAction()` method must take this into account, capitalizing or removing all checked words:

```
  public boolean performActions(MenuItem item) {
    SparseBooleanArray checked=getListView().getCheckedItemPositions();

    switch (item.getItemId()) {
      case R.id.cap:
        for (int i=0; i < checked.size(); i++) {
          if (checked.valueAt(i)) {
            int position=checked.keyAt(i);
            String word=words.get(position);

            word=word.toUpperCase(Locale.ENGLISH);

            adapter.remove(words.get(position));
            adapter.insert(word, position);
          }
        }

        return(true);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      case R.id.remove:
        ArrayList<Integer> positions=new ArrayList<Integer>();

        for (int i=0; i < checked.size(); i++) {
          if (checked.valueAt(i)) {
            positions.add(checked.keyAt(i));
          }
        }

        Collections.sort(positions, Collections.reverseOrder());

        for (int position : positions) {
          adapter.remove(words.get(position));
        }

        getListView().clearChoices();

        return(true);
    }

    return(false);
  }
```

`MultiChoiceModeListener` extends the `ActionMode.Callback` interface we used with our manual action mode earlier in this book. Hence, we need to implement all the standard `ActionMode.Callback` methods, plus a new `onItemCheckedStateChanged()` method introduced by `MultiChoiceModeListener`:

```
package com.commonsware.android.actionmodemc;

import android.annotation.TargetApi;
import android.os.Build;
import android.view.ActionMode;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.AbsListView;
import android.widget.ListView;

@TargetApi(Build.VERSION_CODES.HONEYCOMB)
public class HCMultiChoiceModeListener implements
    AbsListView.MultiChoiceModeListener {
  ActionModeDemo host;
  ActionMode activeMode;
  ListView lv;

  HCMultiChoiceModeListener(ActionModeDemo host, ListView lv) {
    this.host=host;
    this.lv=lv;
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public boolean onCreateActionMode(ActionMode mode, Menu menu) {
    MenuInflater inflater=host.getMenuInflater();

    inflater.inflate(R.menu.context, menu);
    mode.setTitle(R.string.context_title);
    mode.setSubtitle("(1)");
    activeMode=mode;

    return(true);
  }

  @Override
  public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
    return(false);
  }

  @Override
  public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
    boolean result=host.performActions(item);

    updateSubtitle(activeMode);

    return(result);
  }

  @Override
  public void onDestroyActionMode(ActionMode mode) {
    activeMode=null;
  }

  @Override
  public void onItemCheckedStateChanged(ActionMode mode, int position,
                                        long id, boolean checked) {
    updateSubtitle(mode);
  }

  private void updateSubtitle(ActionMode mode) {
    mode.setSubtitle("(" + lv.getCheckedItemCount() + ")");
  }
}
```

Android will automatically start our action mode for us when the user checks the first item in the list, using our `MultiChoiceModeListener` as the callback. Android will also automatically finish the action mode if the user unchecks all previously-checked items.

In `onCreateActionMode()`, we populate the menu, plus set up a title and subtitle on the `ActionMode`. The subtitle appears below the title, as you might expect. In this case, we are indicating how many words are checked and therefore will be affected by the actions the user chooses in the action mode:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

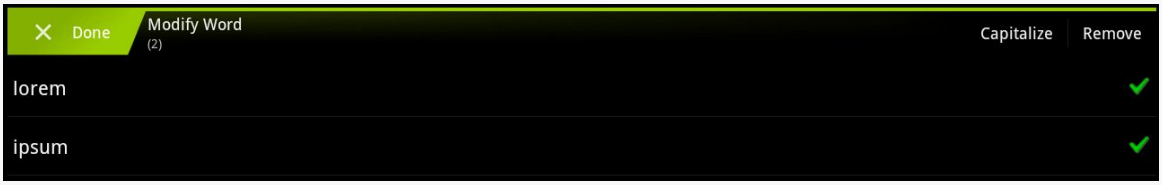


Figure 386: The ActionModeMC sample application's action bar on a Honeycomb tablet, showing the active action mode

Then, in `onActionItemClicked()`, we both call `performActions()` to affect the desired changes, plus update the subtitle in case the user removed words (which means they are no longer checked).

The new `onItemCheckedStateChanged()` will be called whenever the user checks or unchecks an item, up until the last item is unchecked. `HCMultiChoiceModeListener` simply updates the subtitle to reflect the new count of checked items.

On the whole, using `CHOICE_MODE_MULTIPLE_MODAL` is simpler than setting up your own trigger mechanism and managing the action mode yourself. That being said, both are completely valid options, which is particularly important for situations where a multiple-choice `ListView` is not the desired user interface.

Long-Click To Initiate an Action Mode

However, rather than having checkboxes or the like always in the `ListView`, a more modern approach is to move into multiple-selection mode based on a long-click. Before then, clicks on rows behave like with any other `ListView`, but after a long-click, the action mode appears and the user can tap on rows to select which of them to operate upon.

The `ActionMode/LongPress` sample project is a variation on the preceding project, with some slight simplifications, and adopting the long-click as the means to enter the action mode.

Setting Up the Listeners

In `onCreate()`, we set up listeners for both a long click (via `setOnItemLongClickListener()`) and for multiple-choice mode (via `setMultiChoiceModeListener()`. Both times, we supply the activity as the listener, as it implements the appropriate interfaces:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    getListView().setOnItemLongClickListener(this);
    getListView().setMultiChoiceModeListener(this);
```

Handling the Long Click

By default, the `ListView` is in no-choice mode, where clicks on rows simply trigger `onListItemClick()` or the equivalent. However, if the user long-clicks on a row, our `onItemLongClick()` method will be called, and we can both switch into multiple-choice mode and mark the long-clicked row as being checked:

```
  @Override
  public boolean onItemLongClick(AdapterView<?> parent, View view,
                                 int position, long id) {
    getListView().setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
    getListView().setItemChecked(position, true);

    return(true);
  }
```

At this point, the action mode will also start up, courtesy of having called `setMultiChoiceModeListener()`.

Addressing Configuration Changes

If we undergo a configuration change, we want:

1. To keep the current set of words, including any that were added
2. To keep the action mode going, if the user had long-clicked to enter the action mode
3. To keep our checked item states, if the action mode is active

Keeping the checked item states will be handled for us by the built-in instance-state management of `ListView` and `ListActivity`. However, the rest we need to handle ourselves. So, we have an `onSaveInstanceState()` implementation in the activity, which saves the current choice mode, plus the current word list:

```
  @Override
  public void onSaveInstanceState(Bundle state) {
    super.onSaveInstanceState(state);
    state.putInt(STATE_CHOICE_MODE, getListView().getChoiceMode());
    state.putStringArrayList(STATE_MODEL, words);
  }
```

Plus, in `onCreate()`, after setting up the listeners, we set up the choice mode of the `ListView` based upon the passed in instance state `Bundle`, if there is one:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@Override
public void onCreate(Bundle state) {
  super.onCreate(state);

  if (state == null) {
    initAdapter(null);
  }
  else {
    initAdapter(state.getStringArrayList(STATE_MODEL));
  }

  getListView().setOnItemLongClickListener(this);
  getListView().setMultiChoiceModeListener(this);

  int choiceMode=
      (state == null ? ListView.CHOICE_MODE_NONE
          : state.getInt(STATE_CHOICE_MODE));

  getListView().setChoiceMode(choiceMode);
}
```

Once we call `setChoiceMode()` with the previous activity instance's choice mode, if that was `CHOICE_MODE_MULTIPLE_MODAL`, Android will automatically open up the action mode again and restore our checked items.

Resetting the Choice Mode

Where things get a bit interesting is when the user dismisses the action mode, at which point we need to move back to no-choice mode.

You might think that this would merely be a matter of calling `setChoiceMode()` on the `ListView`, asking for `CHOICE_MODE_NONE`. Indeed, that is part of the solution. However, there are two problems:

1. If you call that in `onDestroyActionMode()` directly, you wind up with infinite recursion and a `StackOverflowError`, as changing the choice mode while the action mode is still technically active will cause it to destroy the action mode again.
2. Switching the choice mode back to "none" enables some optimizations within `ListView` that ignore the checked state of our rows. However, those rows still already checked will show up as activated, even after calling `setChoiceMode()` to return to the normal "none" mode. `clearChoices()` also does not have a worthwhile effect, for whatever reason.

Hence, in `onDestroyActionMode()`, not only do we need to call `setChoiceMode()`, but we need to "smack around" the `ListView` enough to get it to clear our checked

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

rows, and the easiest way to do that is to call `setAdapter()` on it, passing in its existing adapter:

```
@Override
public void onDestroyActionMode(ActionMode mode) {
  if (activeMode != null) {
    activeMode=null;
    getListView().setChoiceMode(ListView.CHOICE_MODE_NONE);
    getListView().setAdapter(getListView().getAdapter());
  }
}
```

And, we only do that while our action mode is active (i.e., `activeMode` is not `null`), to avoid the infinite recursion.

This is a bit clunky, but it works.

The Results

When initially launched, the activity looks like a simple `ListActivity`:

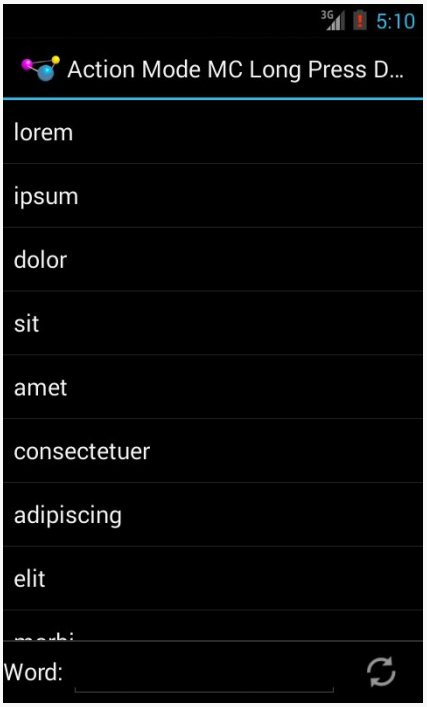


Figure 387: Action Mode Long Press Demo, As Initially Launched

Tapping on a row provides the normal momentary highlight.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, if the user long-clicks a row, we move into the action mode and a multiple-choice `ListView`:

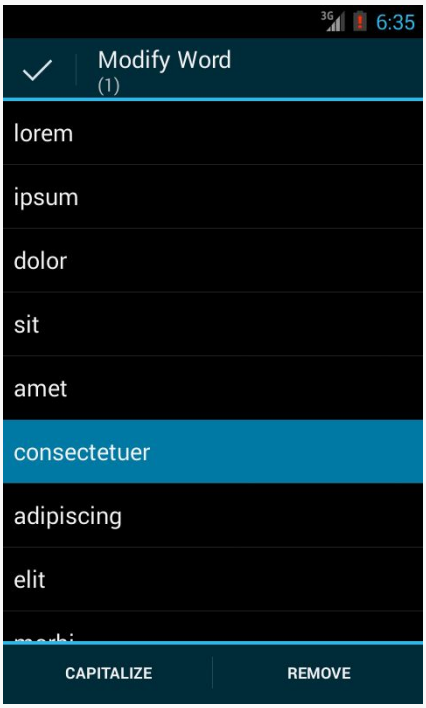


Figure 388: Action Mode Long Press Demo, with Action Mode Activated

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

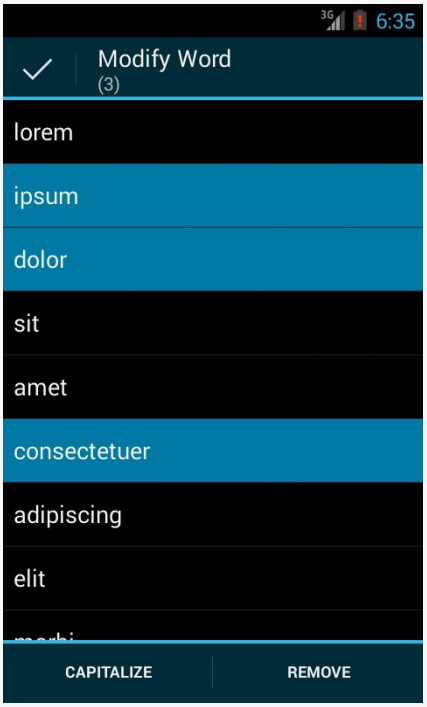


Figure 389: Action Mode Long Press Demo, with Multiple Selections

Dismissing the action mode returns the `ListView` to normal operation.

Split Action Modes

Android 4.0 brought action modes to phone-sized devices. Small screens in the portrait orientation have problems with the action bar in general being too small. Action modes inherit the same problem.

For example, here is the `ActionMode/ActionModeMC` project as seen on a Nexus S running Android 4.0.3:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

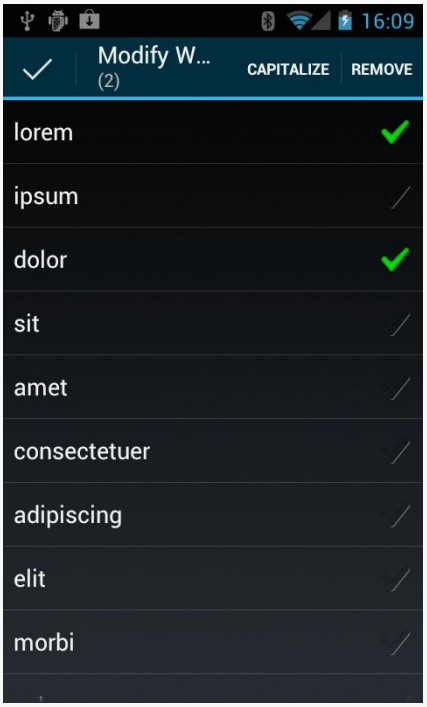


Figure 390: The ActionModeMC sample on a phone

You will notice that our mode's title gets ellipsized due to the lack of room, and this is just with two action items. Admittedly, using icons rather than text labels would help, but even that can only get us so far.

If you use a split action bar, by adding `android:uiOptions="splitActionBarWhenNarrow"` to the `<activity>` element in the manifest, the action mode will also split, with the action items moving to the bottom:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

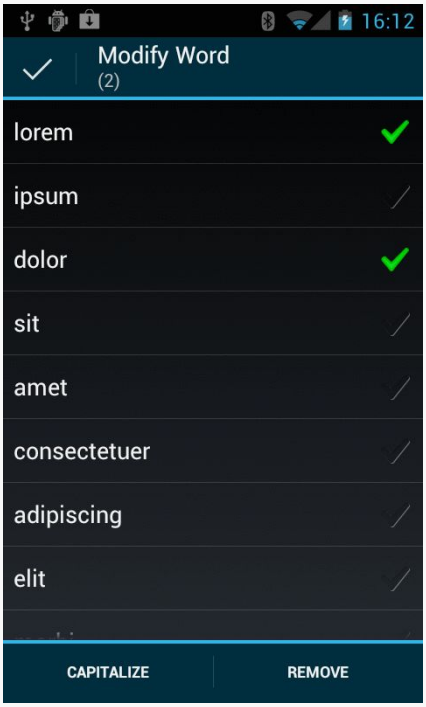


Figure 391: The ActionModeMC sample on a phone, using a split action bar

If there is more horizontal room (i.e., it is not "narrow"), then the action mode will display as normal:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

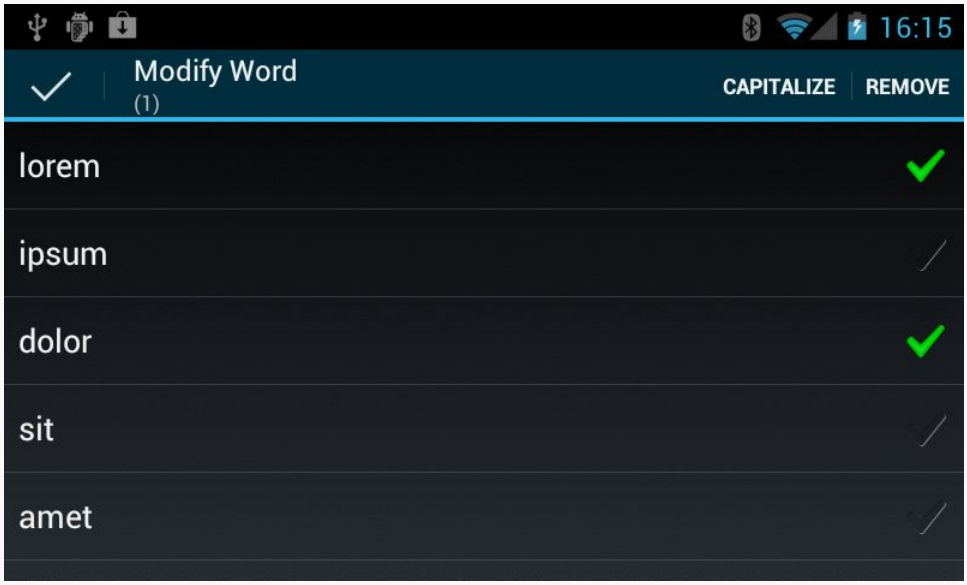


Figure 392: The ActionModeMC sample on a phone, using a split action bar, in landscape orientation

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Other Advanced Action Bar Techniques

The action bar offers a number of other features that developers can take advantage of, ones that do not necessarily fit into the other chapters. Hence, this chapter is a "catch all" for other things you may wish to do with your action bar.

Note that this chapter is focused on the native action bar, not the ActionBarSherlock or AppCompat backports.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on the action bar.

Splitting the Bar

The native action bar debuted with Honeycomb, which was only available for tablets. Here, we had lots of room, even with the device in portrait mode.

Once Ice Cream Sandwich (Android 4.0) rolled around, and the native action bar became available for phones, it was readily apparent that it was too small in portrait mode to do very much.

To help with this, you can enable a mode for your application (or specific activities) that gives you a "split" action bar: one at the top of your activity, and another at the bottom. Your toolbar buttons and the action overflow area will appear at the bottom,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

leaving the top available for your icon, application name, and other stuff that we have not talked about just yet.

To enable this feature, add `android:uiOptions="splitActionBarWhenNarrow"` to your `<application>` or a specific `<activity>` in the manifest. In the sample application manifest shown above, you will see this in the `<application>` element, which means the split action bar is used on *all* activities, by default.

However, this feature is not supported on Android 5.0 and higher, if you are using `Theme.Material` on those devices. The option is simply ignored, and all of your action bar items will go on the main (top) action bar, or into the overflow if there is insufficient room.

For example, the ActionBar/SplitBarDemo sample project is a clone of the `ActionBarDemoNative` application with just one change: adding `android:uiOptions="splitActionBarWhenNarrow"` to the `<application>` element in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.splitbar"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="15"
    android:targetSdkVersion="19"/>

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:uiOptions="splitActionBarWhenNarrow">
    <activity
      android:name="ActionBarDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
</manifest>
```

When using the default `Theme.Holo` on Android 4.x, you get a split action bar, with the action items on the bottom of a "narrow" screen configuration:

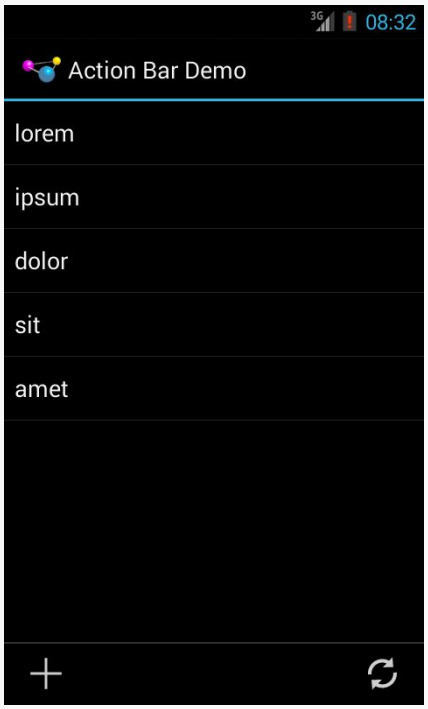


Figure 393: SplitBarDemo on Android 4.3

However, the same sample app, using the default `Theme.Material` on Android 5.0+, will not get the split bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

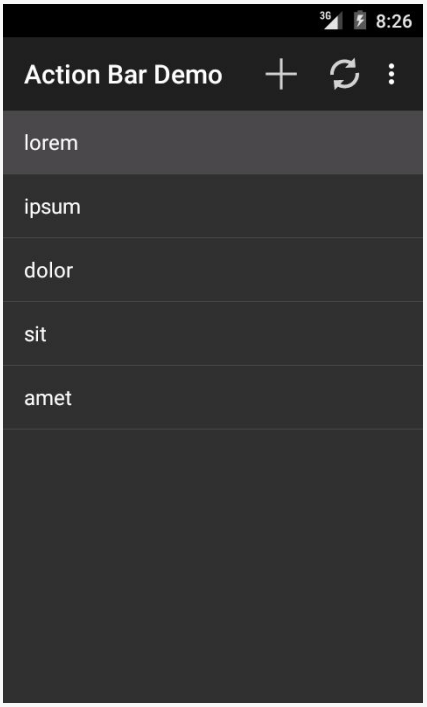


Figure 394: SplitBarDemo on Android 5.0

In this particular sample, everything still fits. However, for apps with more action bar items, we might "lose" some, having them fall into the overflow, whereas they would be directly visible using the split action bar on Android 4.x.

Action Layouts

What happens if you want something other than a button to appear as an action bar item? Suppose you want a field instead?

Fortunately, this is supported. Otherwise, this would be a completely pointless section of the book.

You can specify `android:actionLayout` on an `<item>` element in a menu resource. This will point to a reference to a layout XML resource that you want to have inflated into the action bar instead of a toolbar button. Then, in `onCreateOptionsMenu()`, you can call `findMenuItem()` on the `Menu` to retrieve the `MenuItem` associated with this `<item>` element, then call `getActionView()` to retrieve the root of your inflated layout. At that point, you can hook up event listeners to the widgets in that layout, as needed.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Obviously, since the action bar is only so big, you will need to be judicious about your use of space. This is particularly true on Android 5.0+, when a split action bar is no longer an available option. In practice, you will probably rarely use `android:actionLayout`, if ever.

Floating Action Bars

By default, your action bar (or action bars, if you are using `splitActionBarWhenNarrow`) will be separate from the main content area of your activity. Normally, that is what you want.

But, sometimes, you may want to have the action bar(s) float over the top of your activity, as can be seen in Google Maps:

Figure 395: Google Maps, with Floating Action Bar (image courtesy of Google)

To accomplish this, you can use `FEATURE_ACTION_BAR_OVERLAY`, as is illustrated in the `ActionBar/OverlayNative` sample project.

This is nearly identical to the `ActionBar/ActionBarDemoNative` sample project, with just a few changes, mostly in the `onCreate()` method of our activity:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    getWindow().requestFeature(Window.FEATURE_ACTION_BAR_OVERLAY);

    initAdapter();

    Drawable d=
        getResources().getDrawable(R.drawable.action_bar_background);

    getActionBar().setBackgroundDrawable(d);
    getActionBar().setSplitBackgroundDrawable(d);
  }
```

In addition to the original logic, we:

- Call `requestFeature()` on our `Window` (obtained via a call to `getWindow()`), asking for `FEATURE_ACTION_BAR_OVERLAY`
- Call `setBackgroundDrawable()` on our `ActionBar` (obtained via a call to `getSupportActionBar()`, since we are using ActionBarSherlock, supplying a reference to a drawable resource to use for the background of the floating action bar
- Call `setSplitBackgroundDrawable()` on our `ActionBar` to set the same drawable resource for the background of the bottom action bar, if and when one is used

The drawable resource is a `ShapeDrawable`, defined in XML:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">

  <solid android:color="#AAFFFFFF"/>

</shape>
```

We will discuss `ShapeDrawable` in much greater detail later in this book . For the moment, take it on faith that our resource is defining a rectangle, with a translucent white fill. The alpha channel (`AA`) for our translucence is important, so the user can see a bit of our activity underneath the floating action bar.

The result is that our action bars float over the top of the list:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

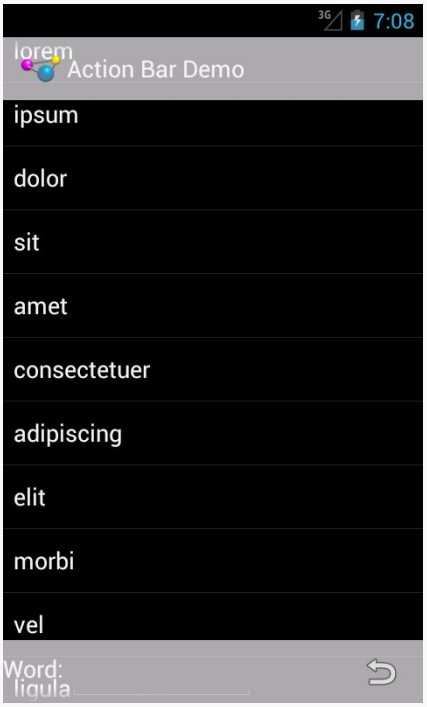


Figure 396: Floating Action Bar

In this case, the effect is not very good, as the words will blend in too strongly with the overlaid action bars. However, that is a question of organizing the screen content and using this overlay feature only in cases where you will see good results, such as in the Google Maps example shown above.

Action Layouts, Action Views, and Action Providers

There are many options for replacing the stock toolbar-style button used for an action item with something else. In the original chapter on the action bar, we saw how to use `android:actionLayout` and `getActionView()` to replace the toolbar button with the contents of a layout resource, which we could then configure at runtime.

If all you need is a single widget to replace the toolbar button, rather than a whole layout resource, you can use `android:actionViewClass` instead of `android:actionLayout`. In `android:actionViewClass`, you provide the fully-qualified class name of the widget that you wish to use to replace the toolbar button. You still use `getActionView()` to retrieve a reference to this at runtime.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If the widget you use implements the `CollapsibleActionView` interface, then it has an additional behavior: the ability to collapse into a standard toolbar button or expand into its normal mode. The only example of this in the current Android SDK is `SearchView`, which can expand into a field for searching or collapse into a simple search icon (magnifying glass) as needed. We will see more about `SearchView`, and how it behaves as a `CollapsibleActionView`, later in this chapter.

Yet another possible toolbar button replacement is an action provider. Whereas an action view or action layout provide the UI, and your code provides the handling of touch events, an action provider is an "all-in-one" solution. It is designed to be configured, then used by the user without any required additional intervention by the developer. That being said, an action provider can have its own listener interfaces to let developers know about various events that have occurred. The two primary implementations of the `ActionProvider` base class are:

- `MediaRouteActionProvider`, used to allow users to control the destination for media, such as routing audio to Bluetooth headphones instead of the device speaker
- `ShareActionProvider`, to simplify sharing content via `ACTION_SEND`, as is covered elsewhere in the book

To use an `ActionProvider`, you add the `android:actionProviderClass` attribute to an `<item>` in the `<menu>` resource, providing the fully-qualified class name of the `ActionProvider` implementation. You can call `getActionProvider()` on the `MenuItem` to retrieve the `ActionProvider` instance, for configuration at runtime.

Searching with SearchView

Many apps employ a `SearchView` in their action bar. The user typically sees the search icon as a regular toolbar button:

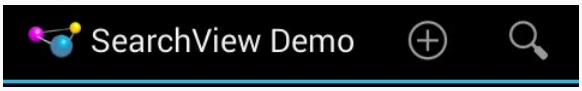


Figure 397: SearchView Demo, Showing Collapsed Action View

Tapping that opens a search field, taking over more of the action bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 398: SearchView Demo, Showing Expanded Action View

Typing something in initiates some sort of search, as defined by the activity that is using the `SearchView`. BACK or the app icon in the action bar will "collapse" the `SearchView` back into its iconified state.

While you are welcome to use `SearchView` in conjunction with Android's overall search framework, you can also use it standalone within your app, as is seen in the `ActionBar/SearchView` sample project, profiled in this section. This sample is a clone of one of the previous action bar samples, where we have the list of 25 words, hosted in a `ListFragment`, with action bar items to add a word and reset the word list. In this section, we will augment the sample with a `SearchView` and a filtered `ListView`.

SearchView... in the Menu Resource

The project's menu resource (`res/menu/actions.xml`) contains a regular action item (`reset`), an action item employing an action layout (`add`), and an action item containing our `SearchView` (`search`):

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/search"
    android:actionViewClass="android.widget.SearchView"
    android:icon="@drawable/ic_action_search"
    android:showAsAction="ifRoom|collapseActionView"
    android:title="@string/filter">
  </item>

</menu>
```

Note that the `search` item not only has `android:actionViewClass="android.widget.SearchView"` to tie in our action view, but it also has `android:showAsAction="ifRoom|collapseActionView"`, to indicate that this action view should support collapsing and expanding.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

SearchView… in the Action Bar Configuration

In `onCreateOptionsMenu()` of our `ActionBarFragment`, in addition to inflating the menu resource and calling a `configureActionItem()` method to configure the `add` action layout, we now also call a `configureSearchView()` method to configure the `SearchView`:

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  inflater.inflate(R.menu.actions, menu);

  configureSearchView(menu);

  super.onCreateOptionsMenu(menu, inflater);
}
```

In `configureSearchView()`, surprisingly enough, we configure the `SearchView`:

```
private void configureSearchView(Menu menu) {
  MenuItem search=menu.findItem(R.id.search);

  sv=(SearchView)search.getActionView();
  sv.setOnQueryTextListener(this);
  sv.setOnCloseListener(this);
  sv.setSubmitButtonEnabled(false);
  sv.setIconifiedByDefault(true);

  if (initialQuery != null) {
    sv.setIconified(false);
    search.expandActionView();
    sv.setQuery(initialQuery, true);
  }
}
```

Specifically, we:

- Register our fragment as the `QueryTextListener` and the `OnCloseListener`, which will be covered in greater detail later in this chapter
- Disable the submit button, as we will be using the `SearchView` for filtering rather than querying
- Indicate that the `SearchView` should be collapsed ("iconified") as the default state

Also, our fragment has an `initialQuery` data member, and if that is not `null`, we expand the `SearchView` and fill in `initialQuery` as the query to be shown in the `SearchView`, also submitting it.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`initialQuery` comes from our configuration change logic, as if the user fills in something in the `SearchView` in one configuration (e.g., portrait), we do not want to lose it on a configuration change (e.g., to landscape). In our `onSaveInstanceState()` method, we save both the query from the `SearchView` and the words currently in our list:

```
  @Override
  public void onSaveInstanceState(Bundle state) {
    super.onSaveInstanceState(state);

    if (!sv.isIconified()) {
      state.putCharSequence(STATE_QUERY, sv.getQuery());
    }

    state.putStringArrayList(STATE_MODEL, words);
  }
```

In `onActivityCreated()`, we use the `savedInstanceState` `Bundle` to populate the adapter with the previous set of words, plus store the old `SearchView`'s query in `initialQuery`:

```
  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);

    if (savedInstanceState == null) {
      initAdapter(null);
    }
    else {
      initAdapter(savedInstanceState.getStringArrayList(STATE_MODEL));
      initialQuery=savedInstanceState.getCharSequence(STATE_QUERY);
    }

    setHasOptionsMenu(true);
  }
```

Hence, on a configuration change, by the time `configureSearchView()` is called, we will have our `initialQuery`, if there is one, and we can set up the UI to be the same as it was in the old configuration.

SearchView… And Filtering a ListView

The `ActionBarFragment` implements the `SearchView.OnQueryTextListener` and `SearchView.OnCloseListener` interfaces, which is why we can pass `this` to `setOnQueryTextListener()` and `setOnCloseListener()` in `configureSearchView()`.

Those two interfaces require a total of three methods, described below.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

onQueryTextChange()

The `onQueryTextChange()` method — required by `SearchView.OnQueryTextListener` – will be called whenever the user has changed the contents of the expanded `SearchView`, such as by typing a character. This is used when you want to employ the `SearchView` for filtering, updating the filter as the user types, rather than for searching, in which case you would wait until the user "submits" the search request.

Our implementation takes advantage of `ArrayAdapter`'s built-in filtering capability:

```
  @Override
  public boolean onQueryTextChange(String newText) {
    if (TextUtils.isEmpty(newText)) {
      adapter.getFilter().filter("");
    }
    else {
      adapter.getFilter().filter(newText.toString());
    }

    return(true);
  }
```

Adapters that implement the `Filterable` interface can be filtered, automatically restricting the displayed items to ones that match the filter. Calling `getFilter()` on a `Filterable` returns a `Filter`. The default implementation of a `Filter` filters on the leading characters of `toString()` of `getItem()` from the `Adapter`. Hence, filtering an `ArrayAdapter` on our roster of 25 words, where the filter string is `'m'`, would show `morbi` and `molllis` but skip `amet`, let alone other words not beginning with `m`.

So, our `onQueryTextChange()` method simply updates the `Filter` with whatever the user has typed into the `SearchView`, setting the filter to the empty string if the `SearchView` is either empty or has `null` contents.

onQueryTextSubmit()

The `onQueryTextSubmit()` method — required by `SearchView.OnQueryTextListener` – would be called if the user tapped on the submit button within the expanded `SearchView`, to ask us to perform the search. In this sample, we have disabled that button, as we are filtering our list on the fly, rather than performing a query once the `SearchView` is filled out. Hence,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`ActionBarFragment` has a do-nothing implementation of `onQueryTextSubmit()`, simply returning `false` to indicate that we have not consumed the event:

```
@Override
public boolean onQueryTextSubmit(String query) {
  return(false);
}
```

The chapter on advanced database techniques has a section on full-text indexing, and the sample app in that chapter demonstrates the use of the submit button in a `SearchView` and `onQueryTextSubmit()`.

onClose()

The `onClose()` method — required by `SearchView.OnCloseListener` — in theory will be called when the `SearchView` is collapsed. Here, we simply clear out the filter that we are using to limit the contents of the `ListView`, plus return `true` to say that we have handled the event:

```
@Override
public boolean onClose() {
  adapter.getFilter().filter("");

  return(true);
}
```

According to the `SearchView` source code, it will only be called if:

- The query text is empty, and
- The `SearchView` is iconified by default (`setIconifiedByDefault(true)`)

In practice, not even that works.

Hence, if you really need to find out when the `SearchView` is collapsed, you will probably need to use the more generic `OnActionExpandListener` interface, attached to the `SearchView` via `setOnActionExpandListener()`. `onMenuItemActionCollapse()` should be called when the `SearchView` is collapsed. This also works for other types of collapsible action views, not just `SearchView`.

SearchView… From the User's Perspective

If the user taps on the search icon, then starts typing into the `SearchView`'s editing area, the `ListView` is filtered based upon the typed-in prefix:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

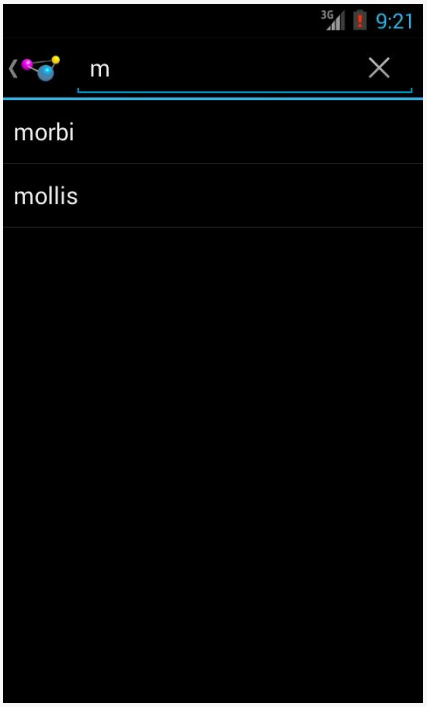


Figure 399: SearchView Demo, Showing Filtered Results

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

AppCompat: The Official Action Bar Backport

Approximately 30 months after Google added the action bar to Android 3.0, Google released a backport for previous devices. Referred to here as AppCompat, this fills much the same niche as does ActionBarSherlock, adding action bar support to Android apps, going all the way back to API Level 7.

The `appcompat-v7` Android Support Package artifact houses AppCompat. Version 21 and higher of this artifact change the way that AppCompat looks, to try to not only backport the action bar, but to backport a bit of the Material Design aesthetic.

This chapter will outline why you might want to use AppCompat and how to employ it in your Android applications.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on the action bar.

Ummmm... Why?

You might wonder why we would bother with any of it. AppCompat is not required, and apps can work fine without it. And, in truth, most apps will be just fine using the native action bar implementation.

That being said, you may find some pressures nudging you towards using an action bar backport, and AppCompat specifically.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Why an Action Bar Backport?

If your `minSdkVersion` is 11 or higher, you have an action bar on all Android versions that your app supports.

If, however, your `minSdkVersion` is below 11, by default you will get the old-style options menu on Android 1.x/2.x devices. That is not a crime. However, the action bar design pattern had been used in various Android apps prior to Android 3.0's formalization of the pattern. Many apps that users will see on older devices will have an action bar, courtesy of one of the backports. By adopting a backport, you will gain a measure of consistency in your UX across Android versions that you would otherwise miss by falling back to the options menu.

You might also adopt a backport because something else is steering you to use AppCompat, and therefore you elect to use it for those reasons.

Why AppCompat?

AppCompat is a somewhat controversial library nowadays. It did not start that way when it was released in the Summer of 2013, other than being the target of "why didn't Google just endorse ActionBarSherlock?" inquiries. But Google has been rather aggressive about trying to get developers to use AppCompat, and that aggressiveness has had its downsides.

Supported

The #1 reason for using AppCompat is because you decided, for other reasons, that you wanted an action bar backport, and AppCompat right now is the primary supported option. ActionBarSherlock is officially deprecated by its author (Jake Wharton), who is steering you towards the native action bar or AppCompat as alternatives. While this does not prevent you from using ActionBarSherlock, it is probably not a great choice today given that Google is supporting AppCompat.

Materialistic

The current version of AppCompat does not only give you an action bar. It gives you an action bar that looks like the one that you get from `Theme.Material`. It also will attempt to apply your accent color to select widgets, the way Android 5.0 and `Theme.Material` do, to make your app abide a bit more by the Material Design aesthetic.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Whether or not this is a good thing is up to you.

Consistent

One hidden advantage of using AppCompat, particularly in concert with the fragments backport, is consistency across Android versions. By using the native action bar and fragments, you are at some risk of inconsistent behavior based upon:

- Android OS version, due to bug fixes, deprecations, and the like
- Manufacturer or ROM modder tweaks to the native implementations, which you do not control

Having your action bar and fragments be in a library in your app isolates you from those changes. In particular, while ActionBarSherlock will defer to the native action bar implementation on API Level 14+ devices, AppCompat always uses its own implementation, so any changes in the native implementation will not affect your app.

This comes at a cost of additional complexity and APK size. Also, while AppCompat is supported, it is not nearly as supported as the native implementations. While it isolates you from inconsistency bugs, it has its own bugs and other issues.

Forced

Some things in the Android development ecosystem, like official support for the `MediaRouteActionProvider`, only work with the AppCompat action bar, as Google has either not shipped or has deprecated their native alternatives.

You may find some "cross-ports" of those things that work with the native action bar, but those are unlikely to be as well-supported as Google's own editions.

While it is theoretically possible that Google itself will eventually offer native action bar implementations of those things, it is unlikely. Hence, if you determine that you need one of those, you may be more inclined to use AppCompat, even if you do not need it for any other reason.

The Basics of Using AppCompat

The recipe for using the AppCompat action bar requires no new skills beyond what you have learned so far in this book. However, there are some subtle and not-so-

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

subtle differences in the approaches AppCompat takes when compared to the native action bar, let alone other backports like ActionBarSherlock.

To see the basic differences, we will take a look at the `AppCompat/ActionBar` sample project. This is a port of the fragments-and-action-bar sample from earlier in the book, where we have replaced the native action bar with AppCompat.

The Library Project

AppCompat is provided by the `appcompat-v7` Android library project, part of the Android Support Package. How you get that depends upon your build environment.

Android Studio and Gradle for Android

Just add the `compile 'com.android.support:appcompat-v7:...'` line to your `dependencies` closure, replacing `...` with a suitable version number of the library. That will take care of downloading the library and adding it to your project.

To get the material effects described in this chapter, you will want to use version 21 or higher of `appcompat-v7` (e.g., `com.android.support:appcompat-v7:21.0.3`).

Eclipse

Your SDK's copy of the AppCompat library project can be found in your Android SDK installation, inside the `extras/android/support/v7/appcompat/` directory.

You will want to import that project into your workspace. Ideally, you should make a copy of the project, so that way when the project gets updated by Google (and you install the update via the SDK Manager), your existing copy is not immediately affected. This will allow *you* to decide when to upgrade to the new AppCompat version.

Your Build Settings

If you are using version 21 or higher of AppCompat, your build target must be API Level 21 or higher. In Android Studio and Gradle for Android, this would be the `compileSdkVersion` found in your `build.gradle` file. In Eclipse, this would be the API level chosen in Project > Properties > Android.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Your Theme

Rather than using `Theme.Holo` or `Theme.Material`, when using AppCompat you will use `Theme.AppCompat`, whether you use that theme directly or create your own custom theme inheriting from it. There is also `Theme.AppCompat.Light` and `Theme.AppCompat.Light.DarkActionBar`, mirroring their native counterparts.

```
  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.AppCompat">
```

Your Menu Resources

Where things start to get a bit strange with AppCompat comes with our menu resources. AppCompat forces you to use a *different namespace* for any action bar-related attributes, those added in API Level 11 or higher.

So, we started with:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/add"
    android:icon="@drawable/ic_action_new"
    android:showAsAction="always"
    android:title="@string/add"/>
  <item
    android:id="@+id/reset"
    android:icon="@drawable/ic_action_refresh"
    android:showAsAction="always|withText"
    android:title="@string/reset"/>

</menu>
```

and we had to change it to:

```
<?xml version="1.0" encoding="utf-8"?>
<menu
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <item
    android:id="@+id/add"
    android:icon="@drawable/ic_action_new"
    app:showAsAction="always"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:title="@string/add"/>
  <item
    android:id="@+id/reset"
    android:icon="@drawable/ic_action_refresh"
    app:showAsAction="always|withText"
    android:title="@string/reset"/>
  <item
    android:id="@+id/about"
    android:icon="@drawable/ic_action_about"
    app:showAsAction="never"
    android:title="@string/about">
  </item>

</menu>
```

Note that we have a new `xmlns:app="http://schemas.android.com/apk/res-auto"` namespace declaration in the root `<menu>` element, and that namespace is used for the `app:showAsAction` attribute. The actual prefix name, here shown as `app`, can be whatever you want. It just has to be unique within the document and a valid XML namespace prefix (e.g., no whitespace).

Your Activity and Fragments

We have to inherit from an `ActionBarActivity` class to use AppCompat. `ActionBarActivity` itself inherits from `FragmentActivity`, and so we can use the Android Support Package's backport of fragments without issue, so you have access to backported versions of `Fragment`, `ListFragment`, etc.

However, note that there are no other analogues of `ActionBarActivity` for other scenarios, such as `ListActivity`. In principle, you should be able to make your own mash-ups of `ActionBarActivity` and other base activity classes, though the proof of this is left as an exercise for the reader. The sample app just uses `ActionBarActivity` directly for showing a `ListView`:

```
package com.commonsware.android.inflation;

import android.os.Bundle;
import android.support.v7.app.ActionBarActivity;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;
import android.widget.ListView;
import android.widget.Toast;
import java.util.ArrayList;

public class ActionBarDemoActivity extends ActionBarActivity {
  private static final String[] items= { "lorem", "ipsum", "dolor",
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
    "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
    "vel", "erat", "placerat", "ante", "porttitor", "sodales",
    "pellentesque", "augue", "purus" };
private ArrayList<String> words=null;
private ArrayAdapter<String> adapter=null;

@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  setContentView(R.layout.list_content_simple);
  initAdapter();
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.actions, menu);

  return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch(item.getItemId()) {
    case R.id.add:
      addWord();

      return(true);

    case R.id.reset:
      initAdapter();

      return(true);

    case R.id.about:
      Toast.makeText(this, R.string.about_toast, Toast.LENGTH_LONG)
           .show();

      return(true);
  }

  return(super.onOptionsItemSelected(item));
}

private void initAdapter() {
  words=new ArrayList<String>();

  for (int i=0;i<5;i++) {
    words.add(items[i]);
  }

  adapter=
      new ArrayAdapter<String>(this,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                                android.R.layout.simple_list_item_1,
                                words);

    setListAdapter(adapter);
  }

  private void addWord() {
    if (adapter.getCount()<items.length) {
      adapter.add(items[adapter.getCount()]);
    }
  }

  private void setListAdapter(ListAdapter la) {
    ((ListView)findViewById(android.R.id.list)).setAdapter(la);
  }
}
```

The layout, `res/layout/list_content_simple.xml`, is cloned from the one used by `ListActivity` itself:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
/* //device/apps/common/assets/res/layout/list_content.xml
**
** Copyright 2006, The Android Open Source Project
**
** Licensed under the Apache License, Version 2.0 (the "License");
** you may not use this file except in compliance with the License.
** You may obtain a copy of the License at
**
**     http://www.apache.org/licenses/LICENSE-2.0
**
** Unless required by applicable law or agreed to in writing, software
** distributed under the License is distributed on an "AS IS" BASIS,
** WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
** See the License for the specific language governing permissions and
** limitations under the License.
*/
-->
<ListView xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@android:id/list"
          android:layout_width="match_parent"
          android:layout_height="match_parent"
          android:drawSelectorOnTop="false"
  />
```

Your Callback Methods

In many cases, your `onCreateOptionsMenu()` and `onOptionsItemSelected()` methods will be the same for AppCompat as they would be for a regular Android app.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, if you do need to manipulate action bar-specific attributes of your inflated menu resources, you will be unable to do so directly. The workaround is to use `MenuCompat` and `MenuItemCompat`, from the Android Support package, to give you access to newer `Menu` and `MenuItem` features in a backwards-compatible fashion.

Your Results

Visually, the results are very similar to what we get from `Theme.Material`, whether we run on Android 5.0 itself:

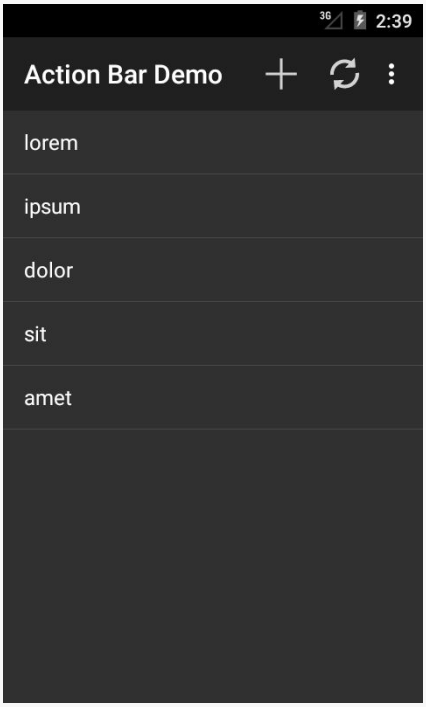


Figure 400: AppCompat, on Android 5.0 Emulator

...or on something older, like an Android 4.1 emulator:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

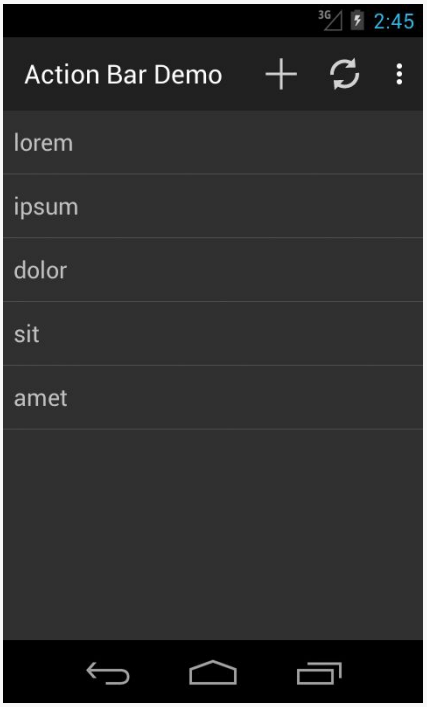


Figure 401: AppCompat, on Android 4.1 Emulator

Other AppCompat Effects

While the above recipe will give you the basics, you can go a lot further with AppCompat, just as you can with the native action bar. Generally speaking, AppCompat's backport includes all of the capabilities of the native action bar.

The biggest key is that when working with `ActionBarActivity`, if you need to access your `ActionBar` instance, call `getSupportActionBar()`, not `getActionBar()`. The latter will compile, but it will return `null` at runtime, as you are disabling the native action bar and using AppCompat's instead.

Tinting

The same basic tinting rules that apply for `Theme.Material` apply for `Theme.AppCompat` and `ActionBarActivity`, with two noteworthy differences.

First, the theme attributes do not have the `android` prefix, but instead are just the bare names (e.g., `colorPrimary`). So, in the `AppCompat/ActionBarColor` sample project, the style resource becomes:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="Theme.Apptheme" parent="Theme.AppCompat">
    <item name="colorPrimary">@color/primary</item>
    <item name="colorPrimaryDark">@color/primary_dark</item>
    <item name="colorAccent">@color/accent</item>
  </style>
</resources>
```

Second, not all widgets will have their colors affected by the theme. At the present time, the roster is limited to:

- `CheckBox`
- `CheckedTextView`
- `EditText`
- `RadioButton`
- `Spinner`

Also, `Switch` adopts the colors, if you use the `SwitchCompat` backport discussed in the next section.

In the `AppCompat/Basic` directory you will find projects mirroring those from the `BasicMaterial` directory. In `BasicMaterial`, we saw how widgets were tinted based on a `Theme.Material`-based theme; in `AppCompat/Basic`, you will see how widgets are tinted based upon a `Theme.AppCompat`-based theme.

Note that AppCompat does not affect the status bar. You can use the Android 5.0 APIs for changing the status bar color on such devices from an `ActionBarActivity`. However, you cannot tint the status bar on older devices, despite using AppCompat.

Switch Backport

As mentioned above, there is an official backport of the `Switch` widget, known as `SwitchCompat`, added to the `appcompat-v7` library. This works back to API Level 7, as does everything in `appcompat-v7`. And, for better or worse, it provides a backport of the Material Design implementation of a `Switch`. So, rather than the `Theme.Holo` ON/OFF toggle, we get the unlabeled "looks like a really tiny `SeekBar`" widget:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

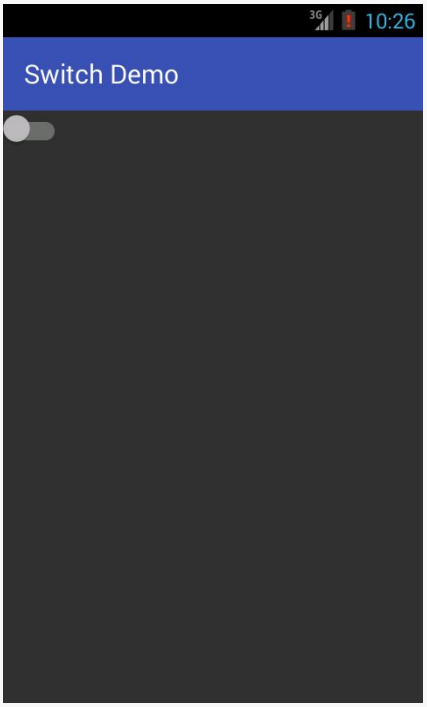


Figure 402: SwitchCompat, on an Android 4.3 Emulator

The above screenshot comes from the AppCompat/Basic/Switch sample project, which uses a layout specifying the `SwitchCompat` widget:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.SwitchCompat
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toggle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Also note that `SwitchCompat` only works inside an `ActionBarActivity`. It is not a general backport of `Switch` that can be used in any activity.

Overlay

AppCompat supports the same basic sort of floating action bar that is supported by the native action bar implementation. There are two slight changes in the recipe:

- Use `supportRequestWindowFeature()`, rather than `requestWindowFeature()`, to request `Window.FEATURE_ACTION_BAR_OVERLAY`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Use `getSupportActionBar()`, rather than `getActionBar()`, to set the background

This is illustrated in `onCreate()` of the `ActionBarDemoActivity` version found in the AppCompat/Overlay sample project:

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  supportRequestWindowFeature(Window.FEATURE_ACTION_BAR_OVERLAY);

  setContentView(R.layout.list_content_simple);

  initAdapter();

  Drawable d=
      getResources().getDrawable(R.drawable.action_bar_background);

  getSupportActionBar().setBackgroundDrawable(d);
  getSupportActionBar().setSplitBackgroundDrawable(d);
}
```

And, you get the same basic results as with the native action bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

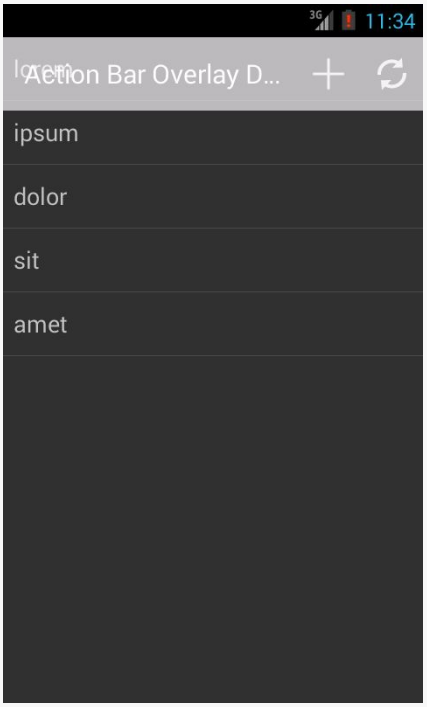


Figure 403: AppCompat with `FEATURE_ACTION_BAR_OVERLAY` on an Android 4.3 Emulator

SearchView

AppCompat has its own implementation of `SearchView`, as `android.support.v7.widget.SearchView`. To use it, switch to that class in your menu resource, then use `MenuItemCompat.getActionView()` to retrieve the instance after your menu resource has been inflated.

For example, the [AppCompat/SearchView] sample project uses the AppCompat implementation of `SearchView` in its `res/menu/actions.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<menu
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <item
    android:id="@+id/search"
    app:actionViewClass="android.support.v7.widget.SearchView"
    android:icon="@drawable/ic_action_search"
    app:showAsAction="ifRoom|collapseActionView"
    android:title="@string/filter">
  </item>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
</menu>
```

Then, in `onCreateOptionsMenu()`, `ActionBarFragment` calls out to a private `configureSearchView()` method that retrieves the `SearchView` and sets it up, much as you saw with the native implementation of `SearchView` from earlier in this book:

```
  private void configureSearchView(Menu menu) {
    MenuItem search=menu.findItem(R.id.search);

    sv=(SearchView)MenuItemCompat.getActionView(search);
    sv.setOnQueryTextListener(this);
    sv.setOnCloseListener(this);
    sv.setSubmitButtonEnabled(false);
    sv.setIconifiedByDefault(true);

    if (initialQuery != null) {
      sv.setIconified(false);
      search.expandActionView();
      sv.setQuery(initialQuery, true);
    }
  }
```

The resulting `SearchView` is tied into AppCompat and offers a Material Design-esque look, applying your tints when opened:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

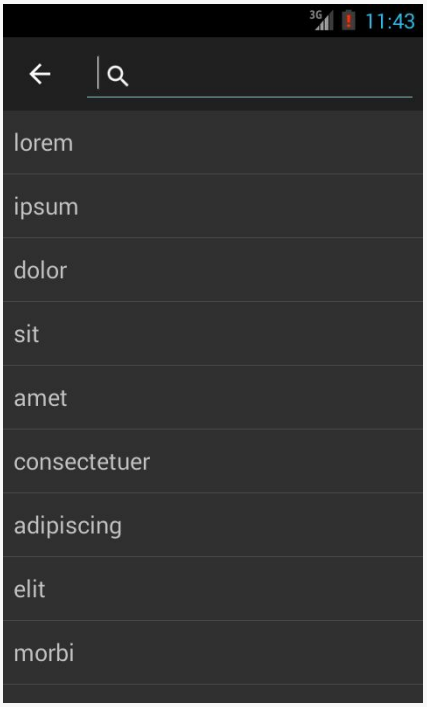


Figure 404: AppCompat with `SearchView` on an Android 4.3 Emulator

ShareActionProvider

Similarly, AppCompat has its own implementation of `ShareActionProvider`, as `android.support.v7.widget.ShareActionProvider`. The recipe for using it resembles that of using `SearchView`:

- Refer to the AppCompat edition of the class in your menu resource
- Use `MenuItemCompat.getActionProvider()` to retrieve your `ShareActionProvider` instance after inflating the menu resource, to configure and use it

The `AppCompat/Share` sample project is a clone of the `ShareActionProvider` project described elsewhere in the book, converted to use AppCompat and its edition of `ShareActionProvider`.

And Then, There Are the Bugs

Unfortunately, AppCompat is not immune to bugs, such as:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Some Samsung devices running Android 4.2 baked in an older AppCompat into their firmware's copy of the zygote, causing class mismatches with apps using the newer AppCompat
- On some HTC devices, some of the ways that AppCompat handles applying its theme break text colors in dialogs

To Material, or Not to Material

(the following is adapted from one of the author's blog posts)

There has been a lot of discussion regarding the adoption of Material Design aesthetics in Android apps. Newcomers to Android might conclude that Material Design must be The Most Important Thing in Android Development and therefore should be pursued immediately at all costs.

Not everybody shares this opinion

With that in mind, here is the author's recommendations on what to consider with Material Design:

DO start considering the effects of Material Design upon your Android app. `Theme.Material` is the dominant theme on Android 5.0 devices, and it should remain the dominant theme on future Android versions, at least for a while. There will be hundreds of millions of these devices in use, eventually, and you will want your app to look like "it belongs" on those devices. On those devices, not only will Google's apps be employing `Theme.Material`, but manufacturer-supplied apps should do so as well. While not everything on the device may necessarily adopt this theme, more than enough will that your app may stand out somewhat if your app does not and the user thinks that it should.

DON'T worry about this right away. Android 4.4 reached 30% of the Android device ecosystem after one year, and we have not seen that rate of adoption since Android 1.0. Odds are that it will be mid–2015 before Android 5.0 crosses the 10% threshold. As such, there just are not that many Android 5.0 devices out there, so worrying about those devices over the next several weeks is serious overkill.

DO plan on including Material Design impacts in your next UI refresh, and you should probably try to budget time for that refresh by the end of 2015 (by which time Android 5.0 might be 20–30% of the ecosystem).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

DON'T blindly assume that you should be using a Material-ish look on all versions of Android. Your objective should be having an app that looks like it belongs on the device. Material-themed apps are highly unlikely to achieve majority status, let alone dominance, on pre-Android 5.0 releases. After all, device manufacturers are not going to be shipping Material-themed updates to built-in apps *en masse*, and there are plenty of apps out there that are still using pre-Holo themes. This is all on top of app that have no identifiable Android theme (e.g., most games). Certainly, there will be Material-themed apps running on Android 4.x devices, courtesy of Google and some other developers. But there will be enough Holo-themed apps that your app will not look out of place on Android 4.x any time soon, if ever. Hence, you have your choice of adopting Material Design across the board or not — user pressure is unlikely to be a major criterion any time soon.

DON'T fall victim to the "our app must look the same across all devices" mindset. Again, your app should look like it belongs on the device, which is why trying to ape an iOS look-and-feel on Android is rarely a good move. Most people do not have two Android devices, and only a small subset of those will have an Android 5.x device and an Android 4.x (or older) device. Hence, most of the people who will care about your app appearing identical on those devices will be in your meeting room discussing the issue. Few of your users will notice — they want an app that works, does what the user wants, and looks like it belongs on their device.

DO consider whether adopting Material Design across the board may offer *engineering* benefits. For example, if you have been heavily customizing widget colors in a `Holo`-based theme, the tinting options provided by `appcompat-v7` may simplify your app a fair bit, reducing APK size, maintenance costs after the Material conversion, etc. Since your users are unlikely to be terribly concerned one way or the other, engineering considerations may help to "tip the scales" in one direction or another.

DON'T blindly assume that `appcompat-v7` will result in a simpler app, or that it should be the basis for all new apps. While the original `appcompat-v7` has now been around for 16 months or so, the new Material `appcompat-v7` is very young, and hence it has bugs. Not all of those bugs are Google's fault (e.g., the Samsung device issue was really caused by a screwy decision on Samsung's part), but that comes as cold comfort to developers trying to distribute `appcompat-v7` apps. And, if you are comfortable with using themes based on `Theme.Holo` for pre-Android 5.0 devices, `appcompat-v7` may be much more of an impediment than an advance. `appcompat-v7` is a tool, not a religion — use it where it clearly adds value *to you and your app*.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

DO start planning for Google's next major theme overhaul. Google, of course, is portraying Material Design as being The One True UI Design. Google will probably get irritated when people point out that `Theme.Material` is the third major theme in the six-year production history of Android, and so assuming that `Theme.Material` is "the be-all and end-all" of themes is unrealistic. Whether the next-generation theme is a refinement on Material Design or a larger overhaul remains to be seen. But you should be taking into account that we may well wind up going through this same process in, say, early 2018. The more you can isolate the theme-related changes from the rest of your app, the more likely it is that you will be able to accommodate future theme changes with less work.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

ActionBarSherlock

As noted in the chapter on AppCompat, there was a ~2.5 year gap where older devices had no official backport of the action bar. Various third-party projects implemented action bars to try to fill this gap, and none did that nearly as well as did ActionBarSherlock.

ActionBarSherlock, in effect, extends the `support-v4` library, adding a backported action bar for apps running on devices prior to API Level 14. While native action bars became available with API Level 11, there were enough differences that ActionBarSherlock uses its own implementation from API Level 13 on down to API Level 7 (Android 2.1).

Today, ActionBarSherlock is considered to be deprecated, with Jake Wharton (the author of ActionBarSherlock) steering developers towards the native action bar or AppCompat. That being said, many existing projects still use ActionBarSherlock successfully today.

To use ActionBarSherlock, you need to do a few things, above and beyond what you would ordinarily need to do to use the native action bar implementation. This chapter reviews what is required and how to use this library.

NOTE: This chapter is scheduled to be removed from this book in one of the 2015 updates.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on the action bar.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Installation

Android Studio users can add a dependency for `com.actionbarsherlock:actionbarsherlock:4.4.0@aar` to their `dependencies` closure. The `@aar` suffix is to help distinguish the AAR artifact from others for this group ID/artifact ID combination.

Eclipse users will need to download ActionBarSherlock, such as by downloading a ZIP file or by cloning the project's GitHub repository. Inside of the ActionBarSherlock distribution is an `actionbarsherlock/` directory, containing an Android library project that you will need to add to your application's project as described in a previous chapter.

Code Changes

There are some changes to your code needed to use ActionBarSherlock, compared with the code you might use with the native API Level 11+ action bar. These changes are fairly minor in scope, to make it easy for you to move back to the native action bar implementation at some time in the future.

Base Activity Class

You will need to adjust your activities to inherit from `SherlockActivity` or one of its kin (e.g., `SherlockListActivity`). This is mostly a matter of adding the `Sherlock` prefix and adjusting your imports to refer to the `com.actionbarsherlock.app` package instead of `android.app`.

This, in turn, will require some changes to the code in the activities themselves:

- References to `Menu` and `MenuItem` need to be changed from the `android.view` package to the `com.actionbarsherlock.view` package
- Calls to `getActionBar()` need to be replaced with `getSupportActionBar()`
- Calls to `getMenuInflater()` need to be replaced with `getSupportMenuInflater()`

Similarly, if you are using fragments, they need to inherit from `SherlockFragment` or some other Sherlock-flavored class (e.g., `SherlockListFragment`). This too will require the aforementioned changes to imports and methods. Also, bear in mind that ActionBarSherlock is using the fragments backport from the Android Support package, and so calls to `getFragmentManager()` need to be replaced by calls to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`getSupportFragmentManager()`, references to classes like `FragmentTransaction` need to be replaced with their Android Support backport equivalents, etc.

Theme

You will also need to apply an ActionBarSherlock-flavored theme to your activities, either on a per-activity basis, or for the application as a whole. The Sherlock theme that most closely resembles the default theme is `Theme.Sherlock`.

The ActionBar/ActionBarDemo sample project applies `Theme.Sherlock` to the whole application, via an `android:theme` attribute on the `<application>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.inflation"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="14"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Sherlock"
    android:uiOptions="splitActionBarWhenNarrow">
    <activity
      android:name=".ActionBarDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

NOTE: If you use this sample app, or any other one that uses ActionBarSherlock, on Eclipse, you will need to update its configuration to point to your own copy of ActionBarSherlock's Android library project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Menu Resources

Your menu resources for use with ActionBarSherlock are largely identical to their native action bar counterparts. In fact, most menu resources will not need to change at all.

The only significant difference is if you are using `ShareActionProvider`. You will need to use a backport of `ShareActionProvider` supplied by the ActionBarSherlock library, `com.actionbarsherlock.widget.ShareActionProvider`. You will need to reference this class in the menu resource, plus use this class in your Java code when you configure the `ShareActionProvider`. By and large, though, the API exposed by the ActionBarSherlock edition of `ShareActionProvider` is the same as the API exposed by the native `ShareActionProvider`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Implementing a Navigation Drawer

Each year brings a new design pattern in Android that takes the development community by storm. In 2011, it was the action bar. In 2012, it was `ViewPager`. In 2013, it was the navigation drawer.

This chapter covers that navigation drawer pattern: what it is, where you use it, and how you implement it, using a `DrawerLayout` class supplied by the Android Support package.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. In addition, one section ties into the chapter on action modes.

What is a Navigation Drawer?

Complex apps often require complex navigation, to get to all of the different areas of the app. And, in many cases, that navigation is tied to nouns, reflecting different types of content, more so than verbs, reflecting operations to be performed against a particular piece of content. Verbs are actions, and can usually go in the action bar as action bar items (e.g., toolbar-style buttons). Nouns *could* be put in the action bar as well as items, though having a mixed bunch of nouns and verbs makes the action bar item roster inconsistent.

Back before the action bar, the "go-to" design pattern for navigation was the so-called "dashboard":

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 405: Google IO 2010 Conference App, with Dashboard

But this took up the whole screen and was therefore only available as the "home" activity of an app.

The navigation drawer, or "sliding menu", pattern has the same sort of content navigation options available in a drawer that slides out from the side of the screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

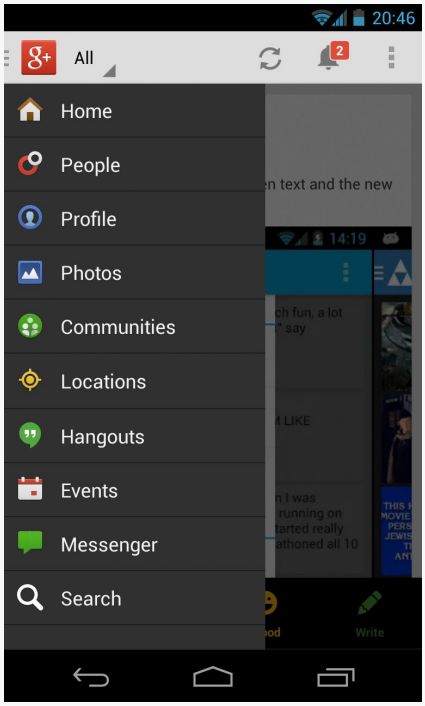


Figure 406: Google+, with Open Navigation Drawer

The drawer can be accessed from many, if not all, activities in the app, to allow the user to get wherever they need to from wherever they happen to be.

A Simple Navigation Drawer

The good news is that Google released an implementation of the navigation drawer pattern, called `DrawerLayout`, in the Android Support package (`support-v4` and `support-v13`).

The bad news is that it still takes a bit of work to get this integrated with your app.

This section will review the `NavDrawer/Simple` sample project, that shows a fairly simplistic integration of a `DrawerLayout` into an activity.

The Activity Layout

The root element of your activity layout, for an activity using `DrawerLayout`, is `DrawerLayout` itself:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<android.support.v4.widget.DrawerLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/drawer_layout"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <FrameLayout
    android:id="@+id/content"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>

  <ListView
    android:id="@+id/drawer"
    android:layout_width="240dp"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    android:background="#111"
    android:choiceMode="singleChoice"
    android:divider="@android:color/transparent"
    android:dividerHeight="0dp"/>

</android.support.v4.widget.DrawerLayout>
```

`DrawerLayout` itself is rather unremarkable in the layout resource: you size and position it, usually to fill the screen. It needs to have precisely two child elements:

1. The first child represents the "real" activity content
2. The second child represents the contents of the drawer that can be opened and closed

If you are adapting an existing activity to use the `DrawerLayout`, that first child could well be an `<include>` element pointing to your existing activity layout resource, so that you can leave it undisturbed and just point your activity to start with this new `DrawerLayout` resource.

There are a couple of attributes in the children that are important for proper `DrawerLayout` operation:

- The second child — often a `ListView` — needs to have its `android:layout_gravity` set to indicate what side of the screen the drawer will slide out from. Typically this will be `left` (or `start` if you are on API Level 17+ and are taking advantage of the RTL layout support).
- The second child also specifies its `android:layout_width` to indicate the size of the drawer when opened. This should *not* be the full width of the screen, and the Android design guidelines suggest a width between 240dp and 320dp.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The ActionBarDrawerToggle

The `onCreate()` method of `MainActivity` is responsible for setting up the drawer, as well as the activity's main content:

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    if (getFragmentManager().findFragmentById(R.id.content) == null) {
      showLorem();
    }

    ListView drawer=(ListView)findViewById(R.id.drawer);

    drawer.setAdapter(new ArrayAdapter<String>(
                                            this,
                                            R.layout.drawer_row,

getResources().getStringArray(R.array.drawer_rows)));
    drawer.setOnItemClickListener(this);

    drawerLayout=(DrawerLayout)findViewById(R.id.drawer_layout);
    toggle=
        new ActionBarDrawerToggle(this, drawerLayout,
                                  R.drawable.ic_drawer,
                                  R.string.drawer_open,
                                  R.string.drawer_close);
    drawerLayout.setDrawerListener(toggle);
    getActionBar().setDisplayHomeAsUpEnabled(true);
    getActionBar().setHomeButtonEnabled(true);
  }
```

In terms of the content, if the `FrameLayout` placed in the layout resource is empty, we call `showLorem()` to lazy-create a `LoremFragment` (a `ListFragment` with 25 Latin words) and run a `FragmentTransaction` to display it:

```
  private void showLorem() {
    if (lorem == null) {
      lorem=new LoremFragment();
    }

    if (!lorem.isVisible()) {
      getFragmentManager().beginTransaction()
                          .replace(R.id.content, lorem).commit();
    }
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`onCreate()` then retrieves the `ListView`, sets the contents of the list to be a `<string-array>` resource named `drawer_rows`, and sets up the activity itself to respond to clicks on the list.

`onCreate()` then sets up an `ActionBarDrawerToggle`. This allows the app icon on the left of the action bar to open and close the navigation drawer. The `ActionBarDrawerToggle` constructor takes five parameters:

- The `Activity` as a `Context`
- The `DrawerLayout` widget to be managed by this toggle
- The icon to superimpose on the app icon to indicate that there is a toggle
- String resources for the open and close operations, for accessibility

Google's "Action Bar Icon Pack" contains stock icons to use for the third parameter, for light and dark action bar themes. In addition, the Android Asset Studio offers a way for you to customize the navigation drawer indicator artwork for other themes.

In addition to creating the toggle instance, we need to:

- Associate it with the `DrawerLayout`, by calling `setDrawerListener()` on it
- Enable the app icon via `setHomeButtonEnabled()` and enable it for "up" navigation via `setDisplayHomeAsUpEnabled()`
- Forward the `onPostCreate()`, `onConfigurationChanged()`, and `onOptionsItemSelected()` activity callback methods on to the toggle:

```
@Override
protected void onPostCreate(Bundle savedInstanceState) {
  super.onPostCreate(savedInstanceState);

  toggle.syncState();
}

@Override
public void onConfigurationChanged(Configuration newConfig) {
  super.onConfigurationChanged(newConfig);

  toggle.onConfigurationChanged(newConfig);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (toggle.onOptionsItemSelected(item)) {
    return(true);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(super.onOptionsItemSelected(item));
  }
```

Without any additional work, launching the app will show the "mini hamburger" navigation drawer icon in the action bar:

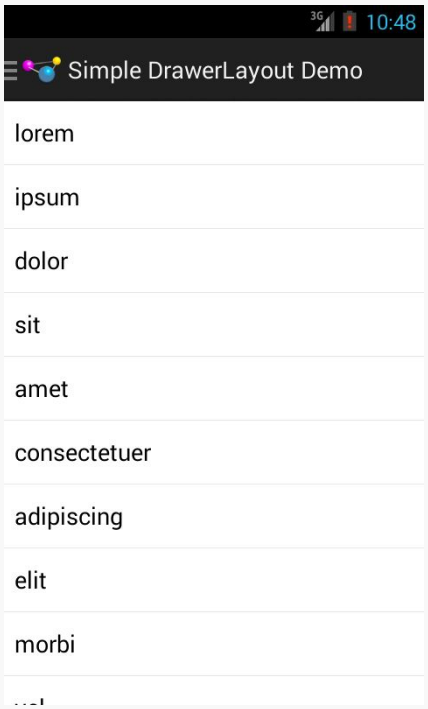


Figure 407: Nav Drawer Sample App, Showing the Mini Hamburger

Tapping the app icon will open the drawer:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

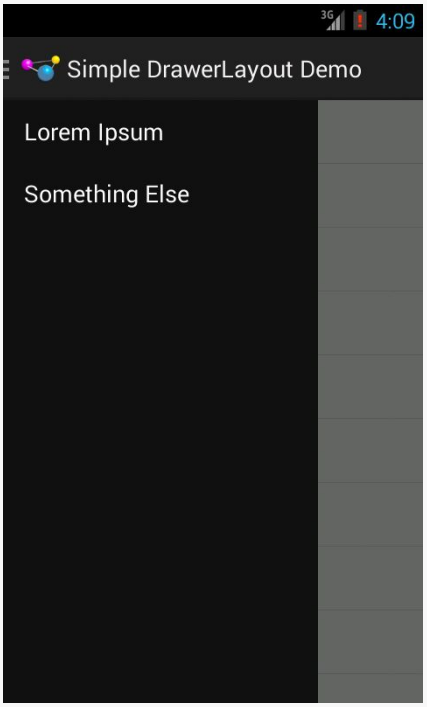


Figure 408: Nav Drawer Sample App, with Drawer Open

The drawer slides over the content, rather than pushing the content away, which is why we do not see the words peeking out on the right side of the screen.

Tapping the app icon again will close the drawer.

The user can also open the drawer via gestures. A bezel swipe from the left side will open the drawer, and swiping the open drawer right to left will close it. In addition, tapping and holding on the left edge of the content will cause the drawer to "peek" open by a handful of pixels, to hint to the user that there may be something that they can access by swiping from the left.

NOTE: This implementation of `ActionBarDrawerToggle` is officially deprecated as of the `21.0.0` version of the Android Support library. However, it is still necessary, at this time, for using `DrawerLayout` with the native action bar.

The Actions on Navigation Clicks

Of course, a navigation drawer is useless unless we do something when the user interacts with it, clicking on list rows in this case.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Our list is very simple, with just two elements. The app simply toggles between two fragments based upon the list item click:

```
@Override
public void onItemClick(AdapterView<?> listView, View row,
                        int position, long id) {
  if (position == 0) {
    showLorem();
  }
  else {
    showContent();
  }

  drawerLayout.closeDrawers();
}
```

We also close the drawer, using `closeDrawers()`, as otherwise the `DrawerLayout` is unaware that the user chose something and that we should return to the content.

Alternative Row Layouts

The rows in the sample app's `ListView` were fairly conventional. Rows in a nav drawer should be fairly simple, as you are merely trying to lead the users to pieces of content, not present content itself.

That being said, the navigation options can have a bit more to them than what the sample app showed. The Android design guidelines will steer you in the direction of how best to style:

- Rows with leading icons
- Rows with trailing badges (e.g., unread message counts)
- Expandable sections (e.g., less-important items but still worth having in the drawer)
- Dividers, to help organize groups of related rows

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

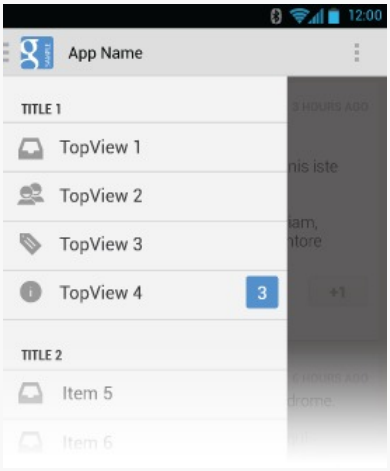


Figure 409: Sample Navigation Drawer Rows with Icons and Badges

You will see other apps experiment with other capabilities. For example, Gmail uses `RadioButton` widgets for the accounts:

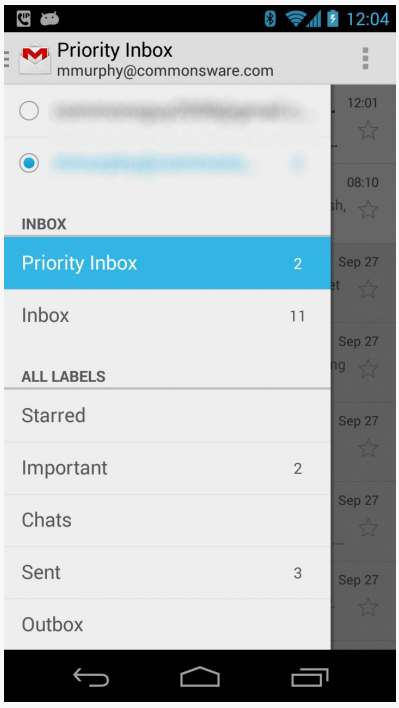


Figure 410: Gmail Navigation Drawer, with RadioButtons, Dividers, and Badges (Sans Backgrounds)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That being said, the closer you can stick with the official design guidelines, the better off you will tend to be, in terms of meeting user expectations and not encountering rendering or click event oddities with `DrawerLayout`.

Additional Considerations

Beyond putting the right things into the navigation drawer, there are some other things that you will need to take into account, particularly as your navigation drawer interacts with the rest of the activity and overall application.

Highlighting the Current Location

If the user opens the navigation drawer, and they are already at one of the navigation destinations shown in the drawer, that destination should show up with the `activated` state, to indicate to the user that she is already there. Conversely, if the user has drilled down into some part of your application that does *not* have a corresponding entry in the navigation drawer, the navigation drawer should show no activated entry.

One way to handle this is to keep the `ListView` updated as the user navigates (whether through the navigation drawer or by other means), selecting and de-selecting items as needed.

The row layout used in the original sample, culled from Google's `DrawerLayout` sample code, already has the `activated` background:

```
<android.support.v4.widget.DrawerLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/drawer_layout"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <FrameLayout
    android:id="@+id/content"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>

  <ListView
    android:id="@+id/drawer"
    android:layout_width="240dp"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    android:background="#111"
    android:choiceMode="singleChoice"
    android:divider="@android:color/transparent"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:dividerHeight="0dp"/>

</android.support.v4.widget.DrawerLayout>
```

However, we were not using it in that sample, so the drawer `ListView` did not give the user any indication that they had already navigated to a navigable destination. But, we have the NavDrawer/Activated sample project, a clone of the original sample, that adds this activation capability, and demonstrates the headaches it causes.

First, as part of our `onCreate()` work in `MainActivity`, we need to configure the `ListView` to work in single-choice mode (the `ListView` engine behind the `activated` state), along with the rest of the `ListView` setup:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  drawer=(ListView)findViewById(R.id.drawer);
  drawer.setChoiceMode(ListView.CHOICE_MODE_SINGLE);

  String[] rows=getResources().getStringArray(R.array.drawer_rows);

  drawer.setAdapter(new ArrayAdapter<String>(this,
                                             R.layout.drawer_row,
                                             rows));
  drawer.setOnItemClickListener(this);

  drawerLayout=(DrawerLayout)findViewById(R.id.drawer_layout);
  toggle=
      new ActionBarDrawerToggle(this, drawerLayout,
                                R.drawable.ic_drawer,
                                R.string.drawer_open,
                                R.string.drawer_close);
  drawerLayout.setDrawerListener(toggle);
  getActionBar().setDisplayHomeAsUpEnabled(true);
  getActionBar().setHomeButtonEnabled(true);

  getFragmentManager().addOnBackStackChangedListener(this);

  if (getFragmentManager().findFragmentById(R.id.content) == null) {
    showLorem();
  }
}
```

Our `showLorem()` method now will `post()` a `Runnable`, named `onNavChange`, after it calls `commit()` on its `FragmentTransaction`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private void showLorem() {
    if (lorem == null) {
      lorem=new LoremFragment();
    }

    if (!lorem.isVisible()) {
      getFragmentManager().popBackStack();
      getFragmentManager().beginTransaction()
                          .replace(R.id.content, lorem).commit();
      drawer.post(onNavChange);
    }
  }
```

That magic `onNavChange` `Runnable` simply sees what fragment is presently visible and updates the checked item in the nav drawer's `ListView` to match:

```
  private Runnable onNavChange=new Runnable() {
    @Override
    public void run() {
      if (lorem != null && lorem.isVisible()) {
        drawer.setItemChecked(0, true);
      }
      else if (content != null && content.isVisible()) {
        drawer.setItemChecked(1, true);
      }
      else {
        int toClear=drawer.getCheckedItemPosition();

        if (toClear >= 0) {
          drawer.setItemChecked(toClear, false);
        }
      }
    }
  };
```

By using `post()`, we schedule the `Runnable` to be executed after the fragment change has occurred. Ideally, we would somehow more explicitly attach the `Runnable` to the `FragmentTransaction`, but that does not appear to be an option.

Since we call `showLorem()` in `onCreate()`, this causes our navigation drawer to show that we are in `LoremFragment` when the activity first starts up:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

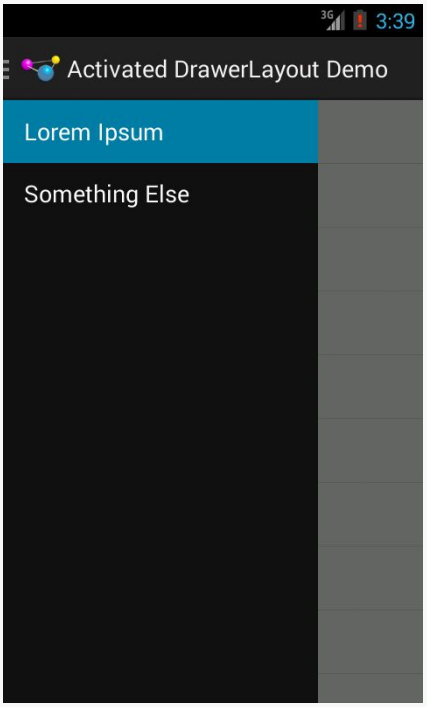


Figure 411: Activated Navigation Drawer Demo, As Initially Launched

The natural behavior of a single-choice `ListView` will check the row that we tap upon, and so when the user taps on an entry in the navigation drawer, it automatically becomes activated. This means we do not have to do our own work for that.

If *everything* in your app is represented by an entry in the navigation drawer, your work is done. However, most likely, there are parts of your app that do not directly map to entries in the navigation drawer... and that is where things get a wee bit complex.

To demonstrate this, we need a bit more to our UI. So, we make `LoremFragment` use the contract pattern, seen elsewhere in the book. We override `onListItemClick()` to call a `wordClicked()` method on the contract, to let the hosting activity know about that UI operation:

```
package com.commonsware.android.drawer.activated;

import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public class LoremFragment extends ContractListFragment<LoremFragment.Contract> {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };

  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);

    setListAdapter(new ArrayAdapter<String>(
                                          getActivity(),
                                          android.R.layout.simple_list_item_1,
                                          items));
  }

  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    getContract().wordClicked();
  }

  interface Contract {
    void wordClicked();
  }
}
```

We also create a `StuffFragment` that displays a simple message:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 412: Activated Navigation Drawer Demo, Showing StuffFragment

The idea is that clicking on a word in the `LoremFragment` should bring up the `StuffFragment`, as if this were a master/detail implementation.

The implementation of `wordClicked()` in `MainActivity` does indeed show a `StuffFragment` (while also adding it to the back stack), but it also needs to make the drawer `ListView` have no checked items, reflecting the fact that the UI state does not reflect one of the navigation destinations:

```
@Override
public void wordClicked() {
  if (stuff == null) {
    stuff=new StuffFragment();
  }

  getFragmentManager().beginTransaction()
                      .replace(R.id.content, stuff)
                      .addToBackStack(null).commit();
  drawer.post(onNavChange);
}
```

Note that we also `post()` the `onNavChange` `Runnable` here as well.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The result is that when we do tap on a word in the list, the `StuffFragment` appears, but the navigation drawer shows no activated row:

Figure 413: Activated Navigation Drawer Demo, Showing StuffFragment and Navigation Drawer

However, since we manually cleared the checked state, we now need to re-check a row if the user navigates back to one of the other fragments. There are two ways the user could return to one of those other fragments: via the navigation drawer, or via the BACK button (popping our transaction off the back stack).

If they navigate to one of the other fragments via the navigation drawer, the appropriate row will be activated automatically by the user's click event. However, we need to consider what to do about that transaction hanging around the back stack from before. One option is to remove it, so the other fragments behave as they do normally, where BACK exits the activity. That is a matter of calling `popBackStack()` on the `FragmentManager` as part of showing one of the fragments, such as the `showContent()` method that shows the `ContentFragment`:

```
  private void showContent() {
    if (content == null) {
      content=new ContentFragment();
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (!content.isVisible()) {
      getFragmentManager().popBackStack();
      getFragmentManager().beginTransaction()
                          .replace(R.id.content, content).commit();
      drawer.post(onNavChange);
    }
  }
```

If the user presses the BACK button, having the earlier fragment reappear happens automatically. However, we need to fix up the navigation drawer to show the proper row as being activated. To do that, we implement `OnBackStackChangedListener` on `MainActivity` and call `addOnBackStackChangedListener()` on the `FragmentManager` in the `onCreate()` initialization work. That way, `onBackStackChanged()` will be called when there is a change in the state of the back stack. Then, it is merely a matter of calling `post()` for `onNavChanged` again, to update the nav drawer:

```
@Override
public void onBackStackChanged() {
  drawer.post(onNavChange);
}
```

Hiding Context-Specific Action Bar Items

Another Google design guideline that makes sense, but adds complexity, is to only show action bar items that pertain to the entire application while the navigation drawer is visible. So, for example, a "Help" action bar item should remain visible, to allow users to switch over to that. But an "Edit" action bar item, to edit something in the main activity, should be hidden while the navigation drawer is visible.

The navigation drawer is effectively an application-level construct, even if we wind up implementing it on a per-activity basis due to the way Android user interfaces are constructed. Hence, the action bar items with the drawer open should pertain to the same scope that the drawer itself does: the application, not a particular activity or fragment inside of it.

Also, on phone-sized screens, the user may not be able to see much of the underlying UI, as the drawer itself will occlude most of it. They may not remember exactly what was showing, and therefore may forget what that "Delete" action bar item would actually delete. Hiding such a context-specific item, while the drawer is open, is safer.

The converse, of course, is that when the drawer closes, you will need to show once again the items that you hid when the drawer opened.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`DrawerLayout` supports a `DrawerListener`, an instance of which you can attach to the drawer itself, to be notified of when the drawer opens and closes, so you can adjust your action bar items to match.

Interacting with an Action Mode

You may be using your own custom action modes (a.k.a., contextual action bars), such as allowing the user to perform operations on multiple selections in a `ListView`.

Since the navigation drawer is tied to the application as a whole, not necessarily just the current activity or the selections in it, Google's recommendation is for you to temporarily dismiss the action mode when the navigation drawer is opened. This will allow the action bar to show items of relevance to the app, not to the selection. If the user navigates elsewhere using the navigation drawer, you would leave the action mode dismissed. If the user slides the navigation drawer closed, though, you could re-enable the action mode, tied back into the multiple selections the user has already made.

Advertising Your Drawer

Users who have spent some time with Android will have a decent shot at recognizing that tapping the action bar item adorned with the nav drawer artwork will open a drawer. However, not all users will necessarily make that connection, particularly users relatively new to Android.

As noted earlier in this chapter, `DrawerLayout` implements the "peek" pattern, where a long-tap on the edge of the screen will cause the drawer to open just a bit, to hint that there is something that can be opened with a swipe gesture. This is nice and subtle, but perhaps too subtle, as users are not necessarily likely to tap-and-hold on the screen edge just to see if something interesting happens.

Another possibility is to have the drawer open automatically, either the first time that your app is launched, or every time your app is launched until you detect that the user has manually opened the drawer (e.g., by adding a `DrawerListener` and watching for `onDrawerSlide()` and `onDrawerOpened()` events not triggered by you). On the plus side, this puts the drawer "front and center", so the user cannot miss it. Once the user taps in your activity (inside or outside of the drawer), the drawer will close with the animated slide to the edge. The hope is that the user will either eventually realize that they can bezel-swipe to get that drawer open, or they will

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

learn about the action bar option (where you have it). This works if the drawer is fairly useful, relative to what the activity would be displaying by default. If, however, the drawer pales in comparison to the main activity view, forcing the drawer open may reduce, not improve, usability.

What Should Not Be in the Drawer

Your navigation drawer should only provide access to the major areas of content in your app. If you do not have many "major areas of content in your app", reconsider having a navigation drawer, and use something else (e.g., `ViewPager` with a tabbed indicator) to get to what content you have.

Also, there are certain things that should not go into a navigation drawer:

- Actions (verbs) generally do not belong in the navigation drawer. "Edit", "Save", and so forth belong as action bar items, or perhaps as part of an action mode, not in the navigation drawer.
- Content itself does not belong in the navigation drawer. Bits of metadata, in the form of badges and similar sorts of indicators, are fine, to hint to the user about the content that is available in that area of your app. But the navigation drawer is not meant to be the "master" in the master/detail pattern, nor is it some sort of "sidebar" of additional content that you just want to have hidden away.
- Low-priority areas of your app, particularly those that are traditionally in the action bar, should not be in the navigation drawer. "Help" and "About" are classic examples. "Settings" is another, though the line starts to get a bit fuzzy (while "Settings" might be something traditional for the action bar, "Accounts" is not). The overflow menu of an action bar is a fine place to have those sorts of areas be available to the user without cluttering up the primary action bar or the navigation drawer.

Independent Implementations

The navigation drawer pattern did not begin with the introduction of `DrawerLayout`. There have been many independent implementations of such a "sliding menu" that pre-dated `DrawerLayout` and the Android design guidelines for navigation drawers. Examples include:

- Jeremy Feinstein's SlidingMenu
- 6Wunderkinder's Sliding Layer

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- David Scott's RibbonMenu

Just bear in mind that these implementations do not necessarily adhere to everything in the design guidelines, requiring you to perhaps make modifications or simply ignore those guidelines as needed.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Advanced Uses of WebView

Android uses the WebKit browser engine as the foundation for both its Browser application and the `WebView` embeddable browsing widget. The Browser application, of course, is something Android users can interact with directly; the `WebView` widget is something you can integrate into your own applications for places where an HTML interface might be useful.

Earlier in this book, we saw a simple integration of a `WebView` into an Android activity, with the activity dictating what the browsing widget displayed and how it responded to links.

Here, we will expand on this theme, and show how to more tightly integrate the Java environment of an Android application with the JavaScript environment of WebKit.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one covering `WebView`.

Friends with Benefits

When you integrate a `WebView` into your activity, you can control what Web pages are displayed, whether they are from a local provider or come from over the Internet, what should happen when a link is clicked, and so forth. And between `WebView`, `WebViewClient`, and `WebSettings`, you can control a fair bit about how the embedded browser behaves. Yet, by default, the browser itself is just a browser, capable of showing Web pages and interacting with Web sites, but otherwise gaining nothing from being hosted by an Android application.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Except for one thing: `addJavascriptInterface()`.

The `addJavascriptInterface()` method on `WebView` allows you to inject a Java object into the `WebView`, exposing its methods, so they can be called by JavaScript loaded by the Web content in the `WebView` itself.

Now you have the power to provide access to a wide range of Android features and capabilities to your `WebView`-hosted content. If you can access it from your activity, and if you can wrap it in something convenient for use by JavaScript, your Web pages can access it as well.

For example, HTML5 offers geolocation, whereby the Web page can find out where the device resides, by browser-supplied means. We can do much of the same thing ourselves via `addJavascriptInterface()`.

In the WebKit/GeoWeb1 project, you will find a fairly simple layout (`main.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <WebView android:id="@+id/webkit"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
  />
</LinearLayout>
```

All this does is host a full-screen `WebView` widget.

Next, take a look at the `GeoWebOne` activity class:

```
package com.commonsware.android.geoweb;

import android.annotation.SuppressLint;
import android.app.Activity;
import android.content.Context;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.os.Bundle;
import android.webkit.JavascriptInterface;
import android.webkit.WebView;
import org.json.JSONException;
import org.json.JSONObject;

public class GeoWebOne extends Activity {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private static String PROVIDER=LocationManager.GPS_PROVIDER;
  private WebView browser;
  private LocationManager myLocationManager=null;

  @SuppressLint("SetJavaScriptEnabled")
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);


myLocationManager=(LocationManager)getSystemService(Context.LOCATION_SERVICE);

    browser.getSettings().setJavaScriptEnabled(true);
    browser.addJavascriptInterface(new Locater(), "locater");
    browser.loadUrl("file:///android_asset/geoweb1.html");
  }

  @Override
  public void onResume() {
    super.onResume();
    myLocationManager.requestLocationUpdates(PROVIDER, 10000,
                                             100.0f,
                                             onLocation);
  }

  @Override
  public void onPause() {
    super.onPause();
    myLocationManager.removeUpdates(onLocation);
  }

  LocationListener onLocation=new LocationListener() {
    public void onLocationChanged(Location location) {
      // ignore...for now
    }

    public void onProviderDisabled(String provider) {
      // required for interface, not used
    }

    public void onProviderEnabled(String provider) {
      // required for interface, not used
    }

    public void onStatusChanged(String provider, int status,
                                Bundle extras) {
      // required for interface, not used
    }
  };

  public class Locater {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    @JavascriptInterface
    public String getLocation() throws JSONException {
      Location loc=myLocationManager.getLastKnownLocation(PROVIDER);

      if (loc==null) {
        return(null);
      }

      JSONObject json=new JSONObject();

      json.put("lat", loc.getLatitude());
      json.put("lon", loc.getLongitude());

      return(json.toString());
    }
  }
}
```

This looks a bit like some of the `WebView` examples from earlier in this book. However, it adds three key bits of code:

- It sets up the `LocationManager` to provide updates when the device position changes, routing those updates to a do-nothing `LocationListener` callback object
- It has a `Locater` inner class that provides a convenient API for accessing the current location, in the form of latitude and longitude values encoded in JSON
- It uses `addJavascriptInterface()` to expose a `Locater` instance under the name `locater` to the Web content loaded in the `WebView`

The Locater API uses JSON to return both a latitude and a longitude at the same time. We are limited to using data types that are in common between JavaScript and Java, so we cannot pass back the `Location` object we get from the `LocationManager`. Hence, we convert the key `Location` data into a simple JSON structure that the JavaScript on the Web page can parse.

Note that the `getLocation()` method on `Locater` has the `@JavascriptInterface` annotation. This is required of apps with `android:targetSdkVersion` set to 17 or higher, though it is a good idea to start using it anyway. With such an `android:targetSdkVersion`, in an app running on an Android 4.2 or higher device, *only* `public` methods with the `@JavascriptInterface` annotation will be accessible by JavaScript code. On earlier devices, or with an earlier `android:targetSdkVersion`, *all* `public` methods on the `Locater` object would be accessible by JavaScript, including those inherited from superclasses like `Object`. Note that your build target

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

will need to be Android 4.2 or higher in order to reference the `@JavascriptInterface` annotation.

Also note that `onCreate()` has the `@SuppressLint("SetJavaScriptEnabled")` annotation. This overrides a Lint warning about the use of `setJavaScriptEnabled(true)`, where Lint wants to make sure that you understand the risks of allowing arbitrary JavaScript to execute inside your app. In this case, the JavaScript is code that we wrote, and so we can ensure that it is safe and sane.

Later in this chapter, we will cover security issues with `WebView` and put those annotations into context.

The Web page itself is referenced in the source code as `file:///android_asset/geoweb1.html`, so the `GeoWeb1` project has a corresponding `assets/` directory containing `geoweb1.html`:

```
<html>
<head>
<title>Android GeoWebOne Demo</title>
<script language="javascript">
  function whereami() {
    var location=JSON.parse(locater.getLocation());

    document.getElementById("lat").innerHTML=location.lat;
    document.getElementById("lon").innerHTML=location.lon;
  }
</script>
</head>
<body>
<p>
You are at: <br/> <span id="lat">(unknown)</span> latitude and <br/>
<span id="lon">(unknown)</span> longitude.
</p>
<p><a onClick="whereami()">Update Location</a></p>
</body>
</html>
```

When you click the “Update Location” link, the page calls a `whereami()` JavaScript function, which in turn uses the `locater` object to update the latitude and longitude, initially shown as “(unknown)” on the page.

If you run the application, initially, the page is pretty boring:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

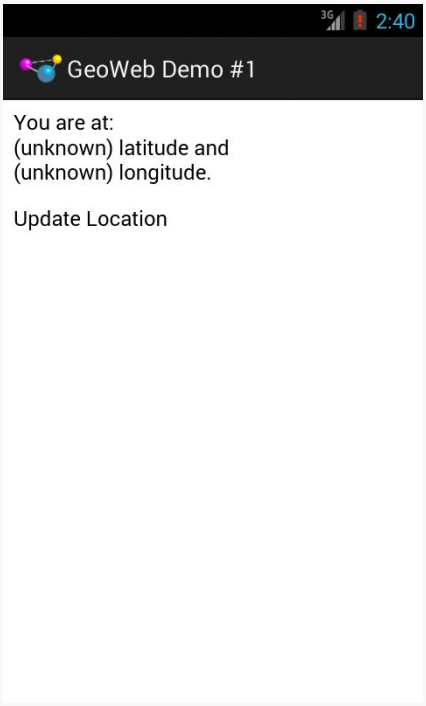


Figure 414: The GeoWebOne sample application, as initially launched

However, if you wait a bit for a GPS fix, and click the "Update Location" link... the page is still pretty boring, but it at least knows where you are:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

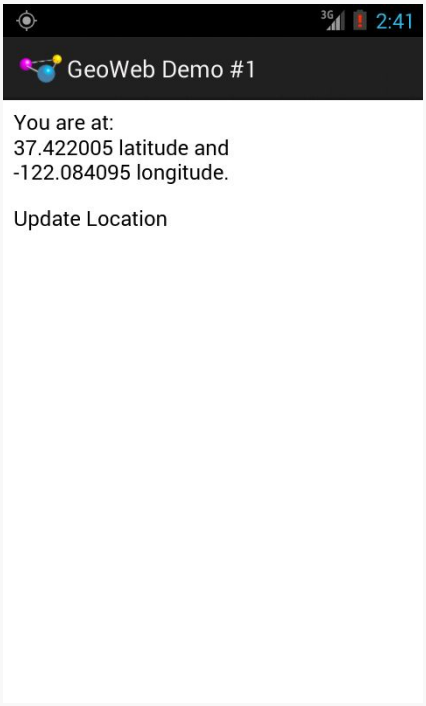


Figure 415: The GeoWebOne sample application, after clicking the Update Location link

Turnabout is Fair Play

Now that we have seen how JavaScript can call into Java, it would be nice if Java could somehow call out to JavaScript. In our example, it would be helpful if we could expose automatic location updates to the Web page, so it could proactively update the position as the user moves, rather than wait for a click on the "Update Location" link.

Well, as luck would have it, we can do that too. This is a good thing, otherwise, this would be a really weak section of the book.

What is unusual is how you call out to JavaScript. One might imagine there would be an `evaluateJavaScript()` counterpart to `addJavascriptInterface()`, where you could supply some JavaScript source and have it executed within the context of the currently-loaded Web page.

Actually, there *is* such a method on Android 4.4. However, earlier versions of Android lacked that method. Instead, on older versions of Android, given your

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

snippet of JavaScript source to execute, you call `loadUrl()` on your `WebView`, as if you were going to load a Web page, but you put `javascript:` in front of your code and use that as the "address" to load.

If you have ever created a "bookmarklet" for a desktop Web browser, you will recognize this technique as being the Android analogue – the `javascript:` prefix tells the browser to treat the rest of the address as JavaScript source, injected into the currently-viewed Web page.

So, armed with this capability, let us modify the previous example to continuously update our position on the Web page.

The layout for the WebKit/GeoWeb2 sample project is the same as before. The Java source for our activity changes a bit:

```
package com.commonsware.android.geoweb2;

import android.annotation.SuppressLint;
import android.annotation.TargetApi;
import android.app.Activity;
import android.content.Context;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.os.Build;
import android.os.Bundle;
import android.webkit.JavascriptInterface;
import android.webkit.WebView;
import org.json.JSONException;
import org.json.JSONObject;

public class GeoWebTwo extends Activity {
  private static String PROVIDER="gps";
  private WebView browser;
  private LocationManager myLocationManager=null;

  @SuppressLint("SetJavaScriptEnabled")
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);

    myLocationManager=
        (LocationManager)getSystemService(Context.LOCATION_SERVICE);

    browser.getSettings().setJavaScriptEnabled(true);
    browser.addJavascriptInterface(new Locater(), "locater");
    browser.loadUrl("file:///android_asset/geoweb2.html");
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@Override
public void onResume() {
  super.onResume();
  myLocationManager.requestLocationUpdates(PROVIDER, 0, 0, onLocation);
}

@Override
public void onPause() {
  super.onPause();
  myLocationManager.removeUpdates(onLocation);
}

LocationListener onLocation=new LocationListener() {
  @TargetApi(Build.VERSION_CODES.KITKAT)
  public void onLocationChanged(Location location) {
    StringBuilder buf=new StringBuilder("whereami(");

    buf.append(String.valueOf(location.getLatitude()));
    buf.append(",");
    buf.append(String.valueOf(location.getLongitude()));
    buf.append(")");

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
      browser.evaluateJavascript(buf.toString(), null);
    }
    else {
      browser.loadUrl("javascript:" + buf.toString());
    }
  }

  public void onProviderDisabled(String provider) {
    // required for interface, not used
  }

  public void onProviderEnabled(String provider) {
    // required for interface, not used
  }

  public void onStatusChanged(String provider, int status,
                              Bundle extras) {
    // required for interface, not used
  }
};

public class Locater {
  @JavascriptInterface
  public String getLocation() throws JSONException {
    Location loc=myLocationManager.getLastKnownLocation(PROVIDER);

    if (loc == null) {
      return(null);
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      JSONObject json=new JSONObject();

      json.put("lat", loc.getLatitude());
      json.put("lon", loc.getLongitude());

      return(json.toString());
    }
  }
}
```

Before, the `onLocationChanged()` method of our `LocationListener` callback did nothing. Now, it builds up a call to a `whereami()` JavaScript function, providing the latitude and longitude as parameters to that call. So, for example, if our location were 40 degrees latitude and –75 degrees longitude, the call would be `whereami(40,-75)`.

What happens then depends upon the version of Android the device is running.

- For devices running Android 4.4+, it calls `evaluateJavascript()`. This takes the JavaScript source code, plus an optional callback, and executes it in the context of the currently-loaded Web page.
- For devices running older versions of Android, it puts `javascript:` in front of the Javascript source and calls `loadUrl()` on the `WebView`. This is the same syntax used for "bookmarklets" in desktop Web browsers.

The result is that a `whereami()` function in the Web page gets called with the new location.

That Web page, of course, also needed a slight revision, to accommodate the option of having the position be passed in:

```
<html>
<head>
<title>Android GeoWebTwo Demo</title>
<script language="javascript">
  function whereami(lat, lon) {
    document.getElementById("lat").innerHTML=lat;
    document.getElementById("lon").innerHTML=lon;
  }

  function pull() {
    var location=JSON.parse(locater.getLocation());

    whereami(location.lat, location.lon);
  }
</script>
</head>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<body>
<p>
You are at: <br/> <span id="lat">(unknown)</span> latitude and <br/>
<span id="lon">(unknown)</span> longitude.
</p>
<p><a onClick="pull()">Update Location</a></p>
</body>
</html>
```

The basics are the same, and we can even keep our "Update Location" link, albeit with a slightly different `onClick` attribute.

If you build, install, and run this revised sample on a GPS-equipped Android device, the page will initially display with "(unknown)" for the current position. After a fix is ready, though, the page will automatically update to reflect your actual position. And, as before, you can always click "Update Location" if you wish.

Navigating the Waters

There is no navigation toolbar with the WebView widget. This allows you to use it in places where such a toolbar would be pointless and a waste of screen real estate. That being said, if you want to offer navigational capabilities, you can, but you have to supply the UI. WebView offers ways to perform garden-variety browser navigation, including:

- `reload()` to refresh the currently-viewed Web page
- `goBack()` to go back one step in the browser history, and `canGoBack()` to determine if there is any history to go back to
- `goForward()` to go forward one step in the browser history, and `canGoForward()` to determine if there is any history to go forward to
- `goBackOrForward()` to go backwards or forwards in the browser history, where negative numbers represent a count of steps to go backwards, and positive numbers represent how many steps to go forwards
- `canGoBackOrForward()` to see if the browser can go backwards or forwards the stated number of steps (following the same positive/negative convention as `goBackOrForward()`)
- `clearCache()` to clear the browser resource cache and `clearHistory()` to clear the browsing history

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Settings, Preferences, and Options (Oh, My!)

With your favorite desktop Web browser, you have some sort of "settings" or "preferences" or "options" window. Between that and the toolbar controls, you can tweak and twiddle the behavior of your browser, from preferred fonts to the behavior of JavaScript.

Similarly, you can adjust the settings of your WebView widget as you see fit, via the `WebSettings` instance returned from calling the widget's `getSettings()` method.

There are lots of options on `WebSettings` to play with. Most appear fairly esoteric (e.g., `setFantasyFontFamily()`). However, here are some that you may find more useful:

- Control the font sizing via `setDefaultFontSize()` (to use a point size) or `setTextSize()` (to use constants indicating relative sizes like LARGER and SMALLEST)
- Control Web site rendering via `setUserAgent()`, so you can supply your own user agent string to make the Web server think you are a desktop browser, another mobile device (e.g., iPhone), or whatever. The settings you change are not persistent, so you should store them somewhere (such as via the Android preferences engine) if you are allowing your users to determine the settings, versus hard-wiring the settings in your application.

Security and Your WebView

More so than normal widgets, `WebView` opens up potential security issues, just as a Web browser could. If all you are doing is displaying your own content, the risks are minimal. If, on the other hand, you are displaying content from third parties, it is possible that their content is malicious in a way that can compromise your app's security, to your users' detriment.

Rogue JavaScript Risks

If you call `setJavaScriptEnabled(true)` on your `WebSettings`, you are allowing JavaScript code to be loaded and executed by `WebView`. In many cases, this is essential to get your content to render properly (e.g., the JavaScript is issuing AJAX calls). However, if you did not write the scripts, you do not know what they might be

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

doing. If there are flaws in `WebView` — such as those discussed in the next sections — then your users may be at risk.

Even in the absence of such bugs, JavaScript can always:

- Consume so much CPU that it represents an attempt at a denial-of-service attack on the user's device
- Access anything the user enters into the Web page
- Access anything *you* enter into the Web page, using approaches as `javascript:` URLs or `evaluateJavaScript()`

The addJavascriptInterface() Bugs

Another way that rogue JavaScript can attack users is if you use `addJavascriptInterface()` to allow JavaScript code to call out to a Java object that you supply.

As was noted earlier in this chapter, when `addJavascriptInterface()` was introduced, there is this `@JavascriptInterface` annotation that we should apply to the methods we want JavaScript to be able to call on the object we supply via `addJavascriptInterface()`. This is because of a bug in the `addJavascriptInterface()` implementation, whereby on 4.1 and below *any* method on the Java object could be called by JavaScript. This includes methods like `getClass()`... which in turn would allow JavaScript to use `Class.forName()` to get at arbitrary stuff. This was used by various bits of malware.

Hence, using `addJavascriptInterface()` on Android 4.1 and below is rather risky, if you are loading arbitrary third-party JavaScript. If you have the means of examining that JavaScript (e.g., you are loading the scripts yourself), you might perform some simple scans of it to see if they appear to be doing anything unfortunate with your Java object that you injected into JavaScript via `addJavascriptInterface()`.

Worse, Android sometimes also injects its *own* objects, without our requesting them.

In particular, this security bug points out that, through Android 4.3, if users have enabled an accessibility service, Android automatically injects objects into `WebView`, using `addJavascriptInterface()`, named `accessibility` and `accessibilityTraversal`. So, even if you do not inject any objects yourself via `addJavascriptInterface()`, your `WebView` may be at risk. The security researchers

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

who uncovered this attack vector suggest using `removeJavascriptInterface()` to specifically get rid of those objects.

The Same-Origin Policy Bug

Due to a bizarre bug in the parsing of URLs, it is possible for JavaScript code to violate the "same-origin policy" of a `WebView` on Android 4.3 and earlier.

Quoting Wikipedia from September 2014:

> the same-origin policy is an important concept in the web application security model. The policy permits scripts running on pages originating from the same site –- a combination of scheme, hostname, and port number –- to access each other's DOM with no specific restrictions, but prevents access to DOM on different sites... This mechanism bears a particular significance for modern web applications that extensively depend on HTTP cookies to maintain authenticated user sessions, as servers act based on the HTTP cookie information to reveal sensitive information or take state-changing actions. A strict separation between content provided by unrelated sites must be maintained on the client side to prevent the loss of data confidentiality or integrity.

All modern Web browsers implement the same-origin policy (SOP)... but there can be bugs. A security researcher disclosed that the original AOSP Browser application failed to implement the SOP properly, when a `javascript:` URL has a null byte before the `j` in `javascript`. And while the report was focused on the AOSP Browser app, the problem really lies with `WebView`.

To see this in action, load `http://commonsware.com/misc/sop-demo.html` in the AOSP Browser app on Android 4.3 or lower. This Web page consists of:

```
<html>
<head>
<title>WebView SOP Test</title>
</head>
<body>
<h1>WebView SOP Test</h1>
<iframe name="test" src="http://developer.android.com"></iframe>
<input type=button value="test"
onclick="window.open('\u0000javascript:alert(document.domain)','test')">
</body>
</html>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It is derived from a similar example found in the blog post outlining the security flaw.

In an SOP-compliant browser, clicking the button will have no effect. In the AOSP Browser app, clicking the button shows the domain name of the document in the `iframe`. And, loading this HTML into a `WebView` has the same effect.

Part of the `WebView` overhaul in Android 4.4 — replacing the original implementation with a new one backed by Chromium — had the effect of fixing this bug, whether intentionally or inadvertently.

There is no obvious mitigation approach for this bug, insofar as for the attack shown above, none of the callbacks on `WebViewClient` or `WebChromeClient` seem to allow us to intercept this URL before its JavaScript is executed. If you are loading HTML yourself from a third party, you might consider scanning that HTML for obvious signs of the attack (e.g., regular expression check for `\u0000javascript`), but that will be limited at best. Beyond that, try to limit the content in a `WebView` to be from only one origin, so that there is nothing for attackers to obtain via this bug.

Also note that the security researcher who found this bug has also found another SOP violation, suggesting that mitigation strategies may be impractical.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Input Method Framework

Android 1.5 introduced the input method framework (IMF), which is commonly referred to as "soft keyboards". However, the "soft keyboard" term is not necessarily accurate, as IMF could be used for handwriting recognition or other means of accepting text input via the screen.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the section covering the `EditText` widget.

Keyboards, Hard and Soft

Some Android devices have a hardware keyboard that is visible some of the time (when it is slid out). A few Android devices have a hardware keyboard that is always visible (so-called "bar" or "slab" phones). Most Android devices, though, have no hardware keyboard at all.

The IMF handles all of these scenarios. In short, if there is no hardware keyboard, an input method editor (IME) will be available to the user when they tap on an enabled `EditText`.

This requires no code changes to your application... if the default functionality of the IME is what you want. Fortunately, Android is fairly smart about guessing what you want, so it may be you can just test with the IME but otherwise make no specific code changes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Of course, the keyboard may not quite behave how you would like. For example, in the `Basic/Field` sample project, the `FieldDemo` activity has the IME overlaying the multiple-line `EditText`:

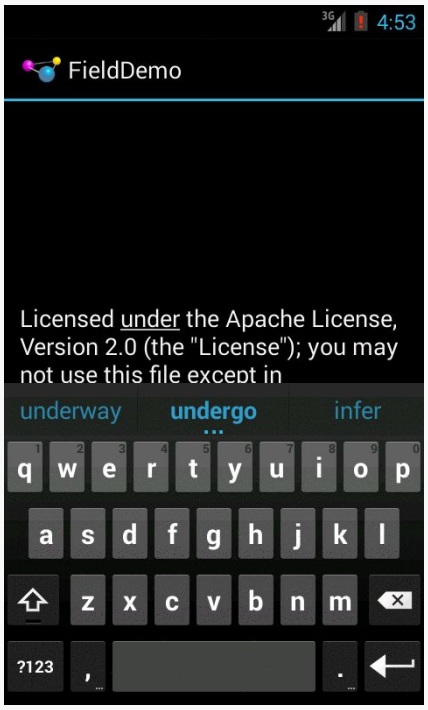


Figure 416: The input method editor, as seen in the FieldDemo sample application

It would be nice to have more control over how this appears, and for other behavior of the IME. Fortunately, the framework as a whole gives you many options for this, as is described over the bulk of this chapter.

Tailored To Your Needs

Android 1.1 and earlier offered many attributes on `EditText` widgets to control their style of input, such as `android:password` to indicate a field should be for password entry (shrouding the password keystrokes from prying eyes). Starting in Android 1.5, with the IMF, many of these have been combined into a single `android:inputType` attribute.

The `android:inputType` attribute takes a class plus modifiers, in a pipe-delimited list (where `|` is the pipe character). The class generally describes what the user is

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

allowed to input, and this determines the basic set of keys available on the soft keyboard. The available classes are:

1. `text` (the default)
2. `number`
3. `phone`
4. `datetime`
5. `date`
6. `time`

Many of these classes offer one or more modifiers, to further refine what the user will be entering. To help explain those, take a look at the `res/layout/main.xml` file from the InputMethod/IMEDemo1 project:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:stretchColumns="1"
  >
  <TableRow>
    <TextView
      android:text="No special rules:"
    />
    <EditText
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Email address:"
    />
    <EditText
      android:inputType="text|textEmailAddress"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Signed decimal number:"
    />
    <EditText
      android:inputType="number|numberSigned|numberDecimal"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Date:"
    />
    <EditText
      android:inputType="date"
    />
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  </TableRow>
  <TableRow>
    <TextView
      android:text="Multi-line text:"
    />
    <EditText
      android:inputType="text|textMultiLine|textAutoCorrect"
      android:minLines="3"
      android:gravity="top"
    />
  </TableRow>
</TableLayout>
```

Here, you will see a `TableLayout` containing five rows, each demonstrating a slightly different flavor of `EditText`:

- One has no attributes at all on the `EditText`, meaning you get a plain text entry field
- One has `android:inputType = "text|textEmailAddress"`, meaning it is text entry, but specifically seeks an email address
- One allows for signed decimal numeric input, via `android:inputType = "number|numberSigned|numberDecimal"`
- One is set up to allow for data entry of a date (`android:inputType = "date"`)
- The last allows for multi-line input with auto-correction of probable spelling errors (`android:inputType = "text|textMultiLine|textAutoCorrect"`)

The class and modifiers tailor the keyboard. So, a plain text entry field results in a plain soft keyboard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

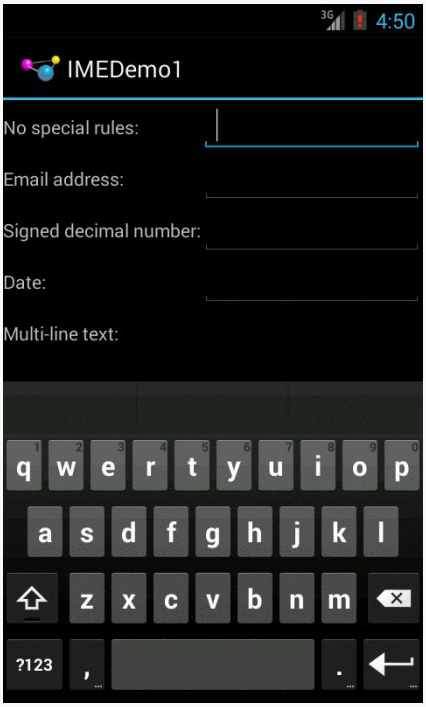


Figure 417: A standard input method editor (a.k.a., soft keyboard)

An email address field might put the @ symbol on the soft keyboard, at the cost of a smaller spacebar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

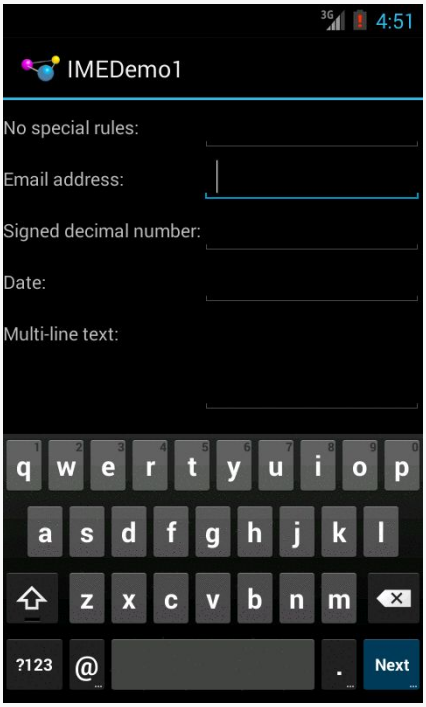


Figure 418: The input method editor for email addresses

Note, though, that this behavior is specific to the input method editor. Some editors might put an @ sign on the primary keyboard for an email field. Some might put a ".com" button on the primary keyboard. Some might not react at all. It is up to the implementation of the input method editor — all you can do is supply the hint.

Numbers and dates restrict the keys to numeric keys, plus a set of symbols that may or may not be valid on a given field:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

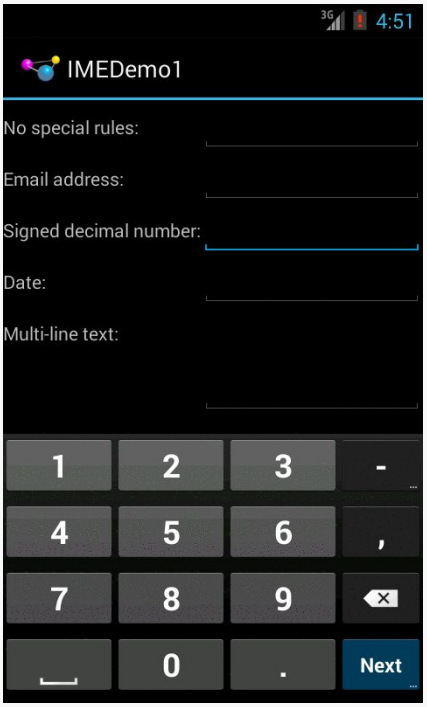


Figure 419: The input method editor for signed decimal numbers

And so on.

By choosing the appropriate `android:inputType`, you can give the user a soft keyboard that best suits what it is they should be entering.

Tell Android Where It Can Go

You may have noticed a subtle difference between the first and second input method editors, beyond the addition of the `@` key. If you look in the lower-right corner of the soft keyboard, the second field's editor has a "Next" button, while the first field's editor has a newline button.

This points out two things:

- `EditText` widgets are multi-line by default if you do not specify `android:inputType`
- You can control what goes on with that lower-right-hand button, called the action key

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

By default, on an `EditText` where you have specified `android:inputType`, the action key will be "Next", moving you to the next `EditText` in sequence, or "Done", if you are on the last `EditText` on the screen. You can manually stipulate what the action key will be labeled via the `android:imeOptions` attribute. For example, in the `res/layout/main.xml` from the InputMethod/IMEDemo2 sample project, you will see an augmented version of the previous example, where two input fields specify what their action key should look like:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
>
  <TableLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="1"
    >
    <TableRow>
      <TextView
        android:text="No special rules:"
      />
      <EditText
      />
    </TableRow>
    <TableRow>
      <TextView
        android:text="Email address:"
      />
      <EditText
        android:inputType="text|textEmailAddress"
        android:imeOptions="actionSend"
      />
    </TableRow>
    <TableRow>
      <TextView
        android:text="Signed decimal number:"
      />
      <EditText
        android:inputType="number|numberSigned|numberDecimal"
        android:imeOptions="actionDone"
      />
    </TableRow>
    <TableRow>
      <TextView
        android:text="Date:"
      />
      <EditText
        android:inputType="date"
      />
    </TableRow>
    <TableRow>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      <TextView
        android:text="Multi-line text:"
      />
      <EditText
        android:inputType="text|textMultiLine|textAutoCorrect"
        android:minLines="3"
        android:gravity="top"
      />
    </TableRow>
  </TableLayout>
</ScrollView>
```

Here, we attach a "Send" action to the action key for the email address (`android:imeOptions = "actionSend"`), and the "Done" action on the middle field (`android:imeOptions = "actionDone"`).

By default, "Next" will move the focus to the next `EditText` and "Done" will close up the input method editor. However, for those, or for any other ones like "Send", you can use `setOnEditorActionListener()` on `EditText` (technically, on the `TextView` superclass) to get control when the action key is clicked or the user presses the `<Enter>` key. You are provided with a flag indicating the desired action (e.g., `IME_ACTION_SEND`), and you can then do something to handle that request (e.g., send an email to the supplied email address).

If you need more control over the action button, you can set:

- `android:imeActionId`, which provides a custom value for the `actionId` that is passed to `onEditorAction()` of your `OnEditorActionListener`
- `android:imeActionLabel`, where you provide your own caption for the button (bearing in mind that your desired caption may or may not fit)

Fitting In

You will notice that the `IMEDemo2` layout shown above has another difference from its `IMEDemo1` predecessor: the use of a `ScrollView` container wrapping the `TableLayout`. This ties into another level of control you have over the input method editors: what happens to your activity's own layout when the input method editor appears?

There are three possibilities, depending on circumstances:

1. Android can "pan" your activity, effectively sliding the whole layout up to accommodate the input method editor, or overlaying your layout, depending

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

on whether the `EditText` being edited is at the top or bottom. This has the effect of hiding some portion of your UI.

2. Android can resize your activity, effectively causing it to shrink to a smaller screen dimension, allowing the input method editor to sit below the activity itself. This is great when the layout can readily be shrunk (e.g., it is dominated by a list or multi-line input field that does not need the whole screen to be functional).
3. In landscape mode, Android may display the input method editor full-screen, obscuring your entire activity. This allows for a bigger keyboard and generally easier data entry.

Android controls the full-screen option purely on its own. And, by default, Android will choose between pan and resize modes depending on what your layout looks like. If you want to specifically choose between pan and resize, you can do so via an `android:windowSoftInputMode` attribute on the `<activity>` element in your `AndroidManifest.xml` file. For example, here is the manifest from `IMEDemo2`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.imf.two"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name=".IMEDemo2"
      android:label="@string/app_name"
      android:windowSoftInputMode="adjustResize">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Because we specified resize, Android will shrink our layout to accommodate the input method editor. With the `ScrollView` in place, this means the scroll bar will appear as needed:

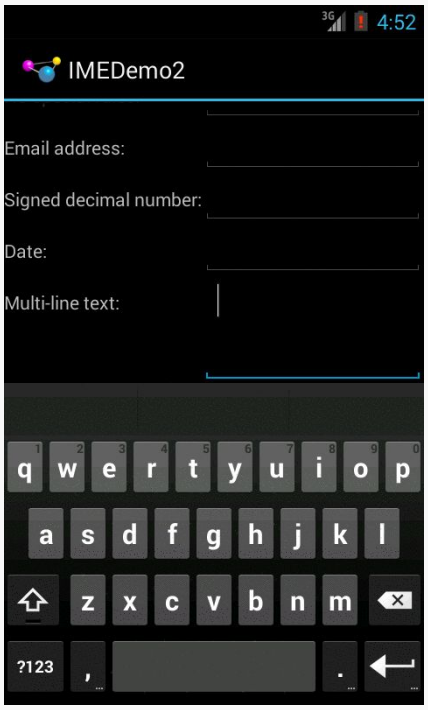


Figure 420: The shrunken, scrollable layout

Jane, Stop This Crazy Thing!

Sometimes, you need the input method editor to just go away. For example, if you make the action button be "Search", the user tapping that button will not automatically hide the editor.

To hide the editor, you will need to make a call to the `InputMethodManager`, a system service that controls these input method editors:

```
InputMethodManager
mgr=(InputMethodManager)getSystemService(INPUT_METHOD_SERVICE);

mgr.hideSoftInputFromWindow(fld.getWindowToken(), 0);
```

(where `fld` is the `EditText` whose input method editor you want to hide)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Fonts

Inevitably, you'll get the question "hey, can we change this font?" when doing application development. The answer depends on what fonts come with the platform, whether you can add other fonts, and how to apply them to the widget or whatever needs the font change.

Android is no different. It comes with some fonts plus a means for adding new fonts. Though, as with any new environment, there are a few idiosyncrasies to deal with.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on files.

Love The One You're With

Android natively knows three fonts, by the shorthand names of "sans", "serif", and "monospace". For Android 1.x, 2.x, and 3.x, these fonts are actually the Droid series of fonts, created for the Open Handset Alliance by Ascender. A new font set, Roboto, is used in Android 4.x and beyond.

For those fonts, you can just reference them in your layout XML, if you choose, such as the following layout from the `Fonts/FontSampler` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:stretchColumns="1">
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <TableRow>
    <TextView
      android:text="sans:"
      android:layout_marginRight="4dip"
      android:textSize="20sp"
    />
    <TextView
      android:id="@+id/sans"
      android:text="Hello, world!"
      android:typeface="sans"
      android:textSize="20sp"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="serif:"
      android:layout_marginRight="4dip"
      android:textSize="20sp"
    />
    <TextView
      android:id="@+id/serif"
      android:text="Hello, world!"
      android:typeface="serif"
      android:textSize="20sp"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="monospace:"
      android:layout_marginRight="4dip"
      android:textSize="20sp"
    />
    <TextView
      android:id="@+id/monospace"
      android:text="Hello, world!"
      android:typeface="monospace"
      android:textSize="20sp"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Custom:"
      android:layout_marginRight="4dip"
      android:textSize="20sp"
    />
    <TextView
      android:id="@+id/custom"
      android:text="Hello, world!"
      android:textSize="20sp"
    />
  </TableRow>
  <TableRow android:id="@+id/filerow">
    <TextView
      android:text="Custom from File:"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      android:layout_marginRight="4dip"
      android:textSize="20sp"
    />
    <TextView
      android:id="@+id/file"
      android:text="Hello, world!"
      android:textSize="20sp"
    />
  </TableRow>
</TableLayout>
```

This layout builds a table showing short samples of five fonts. Notice how the first three have the `android:typeface` attribute, whose value is one of the three built-in font faces (e.g., "sans").

The three built-in fonts are very nice. However, it may be that a designer, or a manager, or a customer wants a different font than one of those three. Or perhaps you want to use a font for specialized purposes, such as a "dingbats" font instead of a series of PNG graphics.

The easiest way to accomplish this is to package the desired font(s) with your application. To do this, simply create an `assets/` folder in the project root, and put your TrueType (TTF) fonts in the assets. You might, for example, create `assets/fonts/` and put your TTF files in there.

Then, you need to tell your widgets to use that font. Unfortunately, you can no longer use layout XML for this, since the XML does not know about any fonts you may have tucked away as an application asset. Instead, you need to make the change in Java code:

```
package com.commonsware.android.fonts;

import android.app.Activity;
import android.graphics.Typeface;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.widget.TextView;
import java.io.File;

public class FontSampler extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    TextView tv=(TextView)findViewById(R.id.custom);
    Typeface face=Typeface.createFromAsset(getAssets(),
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                                              "fonts/HandmadeTypewriter.ttf");

    tv.setTypeface(face);

    File font=new File(Environment.getExternalStorageDirectory(),
                       "MgOpenCosmeticaBold.ttf");

    if (font.exists()) {
      tv=(TextView)findViewById(R.id.file);
      face=Typeface.createFromFile(font);

      tv.setTypeface(face);
    }
    else {
      findViewById(R.id.filerow).setVisibility(View.GONE);
    }
  }
}
```

Here we grab the `TextView` for our "custom" sample, then create a `Typeface` object via the static `createFromAsset()` builder method. This takes the application's `AssetManager` (from `getAssets()`) and a path within your `assets/` directory to the font you want.

Then, it is just a matter of telling the `TextView` to `setTypeface()`, providing the `Typeface` you just created. In this case, we are using the Handmade Typewriter font.

You can also load a font out of a local file and use it. The benefit is that you can customize your fonts after your application has been distributed. On the other hand, you have to somehow arrange to get the font onto the device. But just as you can get a `Typeface` via `createFromAsset()`, you can get a `Typeface` via `createFromFile()`. In our `FontSampler`, we look in the root of "external storage" (typically the SD card) for the MgOpenCosmeticaBold TrueType font file, and if it is found, we use it for the fifth row of the table. Otherwise, we hide that row.

The results?

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

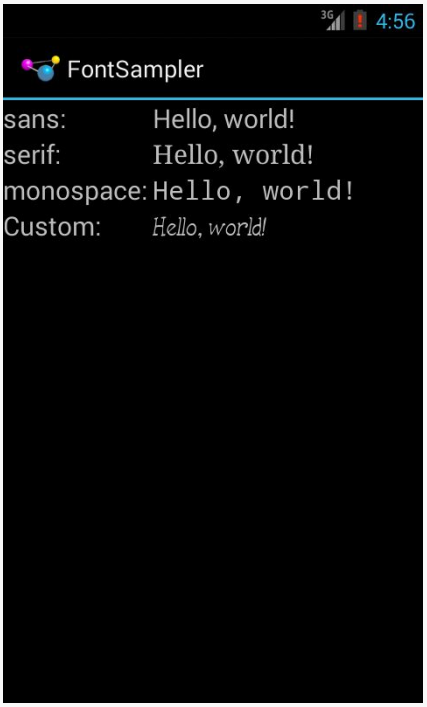


Figure 421: The FontSampler application

Note that Android does not seem to like all TrueType fonts. When Android dislikes a custom font, rather than raise an `Exception`, it seems to substitute Droid Sans ("sans") quietly. So, if you try to use a different font and it does not seem to be working, it may be that the font in question is incompatible with Android, for whatever reason.

Yeah, But Do We Really Have To Do This in Java?

One common complaint with font handling in Android is that you have to apply a custom font on a per-widget basis in Java code.

This gets old quickly.

It is not too bad with just a single `TextView`. But for a whole activity, or a whole application, changing all of the relevant `TextView` widgets (and descendents, like `Button`) gets to be a bit tedious.

While there are "traverse the widget hierarchy and fix up the fonts" code snippets available, you are probably better served using a third-party library, like Christoper

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Jenkins' Calligraphy, which lets you define custom fonts in layout XML files or style resources.

Here a Glyph, There a Glyph

TrueType fonts can be rather pudgy, particularly if they support an extensive subset of the available Unicode characters. The Handmade Typewriter font used above runs over 70KB; the DejaVu free fonts can run upwards of 500KB apiece. Even compressed, these add bulk to your application, so be careful not to go overboard with custom fonts, lest your application take up too much room on your users' phones.

Conversely, bear in mind that fonts may not have all of the glyphs that you need. As an example, let us talk about the ellipsis.

Android's `TextView` class has the built-in ability to "ellipsize" text, truncating it and adding an ellipsis if the text is longer than the available space. You can use this via the `android:ellipsize` attribute, for example. This works fairly well, at least for single-line text.

The ellipsis that Android uses is not three periods. Rather it uses an actual ellipsis character, where the three dots are contained in a single glyph. Hence, any font that you use in a `TextView` where you also use the "ellipsizing" feature will need the ellipsis glyph.

Beyond that, though, Android pads out the string that gets rendered on-screen, such that the length (in characters) is the same before and after "ellipsizing". To make this work, Android replaces one character with the ellipsis, and replaces all other removed characters with the Unicode character 'ZERO WIDTH NO-BREAK SPACE' (`U+FEFF`). This means the "extra" characters after the ellipsis do not take up any visible space on screen, yet they can be part of the string.

However, this means any custom fonts you use for `TextView` widgets that you use with `android:ellipsize` must also support this special Unicode character. Not all fonts do, and you will get artifacts in the on-screen representation of your shortened strings if your font lacks this character (e.g., rogue X's appear at the end of the line).

And, of course, Android's international deployment means your font must handle any language your users might be looking to enter, perhaps through a language-specific input method editor.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Hence, while using custom fonts in Android is very possible, there are many potential problems, and so you must weigh carefully the benefits of the custom fonts versus their potential costs.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Rich Text

Plain text is so, well, plain.

Fortunately, Android has fairly extensive support for formatted text, before you need to break out something as heavy-weight as `WebView`. However, some of this rich text support has been shrouded in mystery, particularly how you would allow users to edit formatted text.

This chapter will explain how the rich text support in Android works and how you can take advantage of it, with particular emphasis on some open source projects to help you do just that.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the ones on basic widgets and the input method framework.

The Span Concept

You may have noticed that many methods in Android accept or return a `CharSequence`. The `CharSequence` interface is little used in traditional Java, if for no other reason than there are relatively few implementations of it outside of `String`. However, in Android, `CharSequence` becomes much more important, because of a sub-interface named `Spanned`.

`Spanned` defines sequences of characters (`CharSequence`) that contain *inline markup rules*. These rules — instances of `CharacterStyle` — indicate whether the "spanned"

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

portion of the characters should be rendered in an alternate font, or be turned into a hyperlink, or have other effects applied to them.

Methods that take a `CharSequence` as a parameter, therefore, can work equally well with `String` objects as well as objects that implement `Spanned`.

Implementations

The base interface for rich-text `CharSequence` objects is `Spanned`. This is used for any `CharSequence` that has inline markup rules, and it defines methods for retrieving markup rules applied to portions of the underlying text.

The primary concrete implementation of `Spanned` is `SpannedString`. `SpannedString`, like `String`, is immutable — you cannot change either the text or the formatting of a `SpannedString`.

There is also the `Spannable` sub-interface of `Spanned`. `Spannable` is used for any `CharSequence` with inline markup rules that can be modified, and it defines the methods for modifying the formatting. There is a corresponding `SpannableString` implementation.

Finally, there is a related `Editable` interface, which is for a `CharSequence` that can have its *text* modified in-place. `SpannableStringBuilder` implements both `Editable` and `Spannable`, for modifying text and formatting at the same time.

TextView and Spanned

One of the most important uses of `Spanned` objects is with `TextView`. `TextView` is capable of rendering a `Spanned`, complete with all of the specified formatting. So, if you have a `Spanned` that indicates that the third word should be rendered in italics, `TextView` will faithfully italicize that word.

`TextView`, of course, is an ancestor of many other widgets, from `EditText` to `Button` to `CheckBox`. Each of those, therefore, can use and render `Spannable` objects. The fact that `EditText` has the ability to render `Spanned` objects — and even allow them to be edited — is key for allowing users to enter rich text themselves as part of your UI.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Available Spans

As noted above, the markup rules come in the form of instances of a base class known as `CharacterStyle`. Despite that name, all of the SDK-supplied subclasses of `CharacterStyle` end in `Span` (not `Style`), and so you will likely see references to these as "spans" as often as "styles". That also helps minimize confusion between character styles and style resources.

There are well over a dozen supplied `CharacterStyle` subclasses, including:

1. `ForegroundColorSpan` and `BackgroundColorSpan` for coloring text
2. `StyleSpan`, `TextAppearanceSpan`, `TypefaceSpan`, `UnderlineSpan`, and `StrikethroughSpan` for affecting the true "style" of text
3. `AbsoluteSizeSpan`, `RelativeSizeSpan`, `SuperscriptSpan`, and `SubscriptSpan` for affecting the size (and, in some cases, vertical position) of the text

And so on.

In principle, you could implement your own custom subclasses of `CharacterStyle`, though coverage of this is well outside the scope of this book.

Loading Rich Text

`Spanned` objects do not appear by magic. Plenty of things in Java will give you ordinary strings, from XML and JSON parsers to loading data out of a database to simply hard-coding string constants. However, there are only a few ways that you as a developer will get a `Spanned` complete with formatting, and that includes you creating such a `Spanned` yourself by hand.

String Resource

The primary way most developers get a `Spanned` object into their application is via a string resource. String resources support inline markup in the form of HTML tags. Bold (`<b>`), italics (`<i>`), and underline (`<u>`) are officially supported, such as:

```
<string name="welcome">Welcome to <b>Android</b>!</string>
```

When you retrieve the string resource via `getText()`, you get back a `CharSequence` that represents a `Spanned` object with the markup rules in place.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

HTML

The next-most common way to get a `Spanned` object is to use `Html.fromHtml()`. This parses an HTML string and returns a `Spanned` object, with all recognized tags converted into corresponding spans. You might use this for text loaded from a database, retrieved from a Web service call, extracted from an RSS feed, etc.

Unfortunately, the list of tags that `fromHtml()` understands is undocumented. Based upon the source code to `fromHtml()`, the following seem safe:

- `<a href="...">`
- `<b>`
- `<big>`
- `<blockquote>`
- `<br>`
- `<cite>`
- `<dfn>`
- `<div align="...">`
- `<em>`
- `<font size="..." color="..." face="...">`
- `<h1>`
- `<h2>`
- `<h3>`
- `<h4>`
- `<h5>`
- `<h6>`
- `<i>`
- `<img src="...">`
- `<p>`
- `<small>`
- `<strike>`
- `<strong>`
- `<sub>`
- `<sup>`
- `<tt>`
- `<u>`

However, do bear in mind that these are undocumented and therefore are subject to change. Also note that `fromHtml()` is perhaps slower than you might think, particularly for longer strings.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You might also wind up using some other support code to get your HTML. For example, some data sources might publish text formatted as Markdown — Stack Overflow, GitHub, etc. use this extensively. Markdown can be converted to HTML, through any number of available Java libraries or via CWAC-AndDown, which wraps the native hoedown Markdown-to-HTML converter for maximum speed. CWAC-AndDown will be explored in a bit more detail in the chapter on the NDK.

From EditText

The reason why so much sample code calls `getText()` followed by `toString()` on an `EditText` widget is because `EditText` is going to return an `Editable` object from `getText()`, not a simple string. That's because, in theory, `EditText` could be returning something with formatting applied. The call to `toString()` simply strips out any potential formatting as part of giving you back a `String`.

However, you could elect to use the `Editable` object (presumably a `SpannableStringBuilder`) if you wanted, such as for pouring the entered text into a `TextView`, complete with any formatting that might have wound up on the entered text.

Actually getting formatting applied to the contents of an `EditText` is covered later in this chapter.

Manually

You are welcome to create a `SpannableString` via its constructor, supplying the text that you wish to display, then calling various methods on `SpannableString` to format it. We will see an example of this later in this chapter.

Or, you are welcome to create a `SpannableStringBuilder` via its constructor. In some respects, `SpannableStringBuilder` works like the classic `StringBuilder` — you call `append()` to add more text. However, `SpannableStringBuilder` also offers `delete()`, `insert()`, and `replace()` methods to modify portions of the existing content. It also supports the same methods that `SpannableString` does, via the `Spannable` interface, for applying formatting rules to portions of text.

Editing Rich Text

If the `Spannable` you wound up with is a `SpannedString`, it is what it is — you cannot change it. If, however, you have a `SpannableString`, that can be modified by

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

you, or by the user. Of course, allowing the user to modify a `Spannable` gets a wee bit tricky, and is why the `RichEditText` project was born.

RichEditText

If you load a `Spannable` into an `EditText`, the formatting will not only be displayed, but it will be part of the editing experience. For example, if the phrase "**the fox jumped**" is in bold, and the user adds in more words to make it "**the quick brown fox jumped**", the additional words will also be in boldface. That is because the user is modifying text in the middle of a defined span, and so therefore the adjusted text is rendered according to that span.

The biggest problem is that `EditText` alone has no mechanism to allow users to *change* formatting. Perhaps someday it will have options for that. In the meantime, though, `RichEditText` is designed to fill that gap.

`RichEditText` is a CWAC project that offers a reasonably convenient API for applying, toggling, or removing effects applied to the current selected text. You have your choice of creating your own UI for this (e.g., implementing a toolbar) or enabling an extension to the `EditText` action modes to allow the users to format the text.

More information on using `RichEditText` can be found on the project site, and a future version of this chapter will go into details not only of its use, but also its construction, once the project has matured a little more.

Manually

`Spannable` offers two methods for modifying its formatting: `setSpan()` to apply formatting, and `removeSpan()` to get rid of an existing span. And, since `Spannable` extends `Spanned`, a `Spannable` also has `getSpans()`, to return existing spans of a current type within a certain range of characters in the text. These methods, along with others on `Spanned`, allow you to get and set whatever formatting you wish to apply on a `Spannable` object, such as a `SpannableString`.

For example, let's take a look at the `RichText/Search` sample project. Here, we are going to load some text into a `TextView`, then allow the user to enter a search string in an `EditText`, and we will use the `Spannable` methods to highlight the search string occurrences inside the text in the `TextView`.

Our layout is simply an `EditText` atop a `TextView` (wrapped in a `ScrollView`):

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <EditText
    android:id="@+id/search"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:singleLine="true">

    <requestFocus/>
  </EditText>

  <ScrollView
    android:id="@+id/scroll"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
      android:id="@+id/prose"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="@string/address"
      android:textAppearance="?android:attr/textAppearanceMedium"/>
  </ScrollView>

</LinearLayout>
```

We pre-fill the `TextView` with a string resource (`@string/address`), which in this project is the text of Lincoln's Gettysburg Address, with a bit of inline markup (e.g., "Four score and seven years ago" italicized). So, when we fire up the project at the outset, we see the formatted prose from the string resource:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

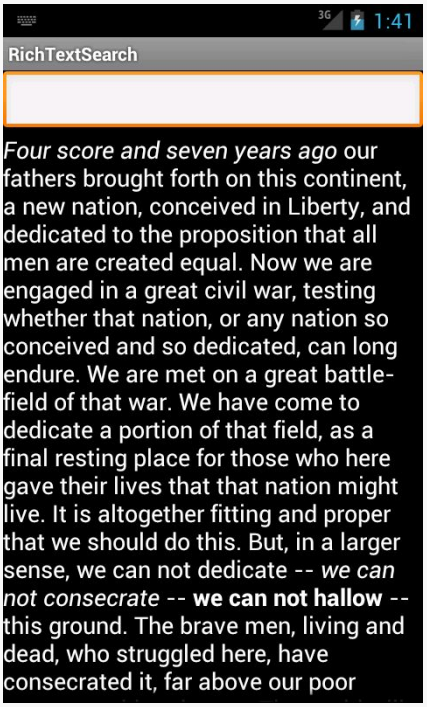


Figure 422: The RichTextSearch sample, as initially launched

In `onCreate()` of our activity, we find the `EditText` widget and designate the activity itself as being an `OnEditorActionListener` for the `EditText`:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    search=(EditText)findViewById(R.id.search);
    search.setOnEditorActionListener(this);
  }
```

That means when the user presses <Enter>, we will get control in an `onEditorAction()` method. There, we pass the search text to a private `searchFor()` method, plus ensure that the input method editor is hidden (if one was used to fill in the search text):

```
  @Override
  public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {
    if (event == null || event.getAction() == KeyEvent.ACTION_UP) {
      searchFor(search.getText().toString());

      InputMethodManager imm=
          (InputMethodManager)getSystemService(INPUT_METHOD_SERVICE);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      imm.hideSoftInputFromWindow(v.getWindowToken(), 0);
    }

    return(true);
  }
```

The `searchFor()` method is where the formatting is applied to our search text:

```
  private void searchFor(String text) {
    TextView prose=(TextView)findViewById(R.id.prose);
    Spannable raw=new SpannableString(prose.getText());
    BackgroundColorSpan[] spans=raw.getSpans(0,
                                             raw.length(),
                                             BackgroundColorSpan.class);

    for (BackgroundColorSpan span : spans) {
      raw.removeSpan(span);
    }

    int index=TextUtils.indexOf(raw, text);

    while (index >= 0) {
      raw.setSpan(new BackgroundColorSpan(0xFF8B008B), index, index
          + text.length(), Spanned.SPAN_EXCLUSIVE_EXCLUSIVE);
      index=TextUtils.indexOf(raw, text, index + text.length());
    }

    prose.setText(raw);
  }
```

First, we get a `Spannable` object out of the `TextView`. While an `EditText` returns an `Editable` from `getText()`, `getText()` on a `TextView` returns a `CharSequence`. In particular, the first time we execute `searchFor()`, `getText()` will return a `SpannedString`, as that is what a string resource turns into. However, that is not modifiable, so we convert it into a `SpannableString` so we can apply formatting to it. An optimization would be to see if `getText()` returns something implementing `Spannable` and then just using it directly.

We want to highlight the search terms using a `BackgroundColorSpan`. However, that means we first need to get rid of any existing `BackgroundColorSpan` objects applied to the prose from a previous search — otherwise, we would keep highlighting more and more of the prose. So, we use `getSpans()` to find all `BackgroundColorSpan` objects anywhere in the prose (from index 0 through the length of the text). For each that we find, we call `removeSpan()` to get rid of it from our `Spannable`.

Then, we use `indexOf()` on `TextUtils` to find the first occurrence of whatever the user typed into the `EditText`. If we find it, we create a new `BackgroundColorSpan`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

and apply it to the matching portion of the prose using `setSpan()`. The last parameter to `setSpan()` is a flag, indicating what should happen if text is inserted at either the starting or ending point. In our case, the text itself is remaining constant, so the flag does not matter much – here, we use `SPAN_EXCLUSIVE_EXCLUSIVE`, which would mean that the span would not cover any text inserted at the starting or ending point of the span.

We then continue using `indexOf()` to find any remaining occurrences of the search text. Once we are done modifying our `Spannable`, we put it into the `TextView` via `setText()`.

The result is that all matching substrings are highlighted in a purple/magenta shade:

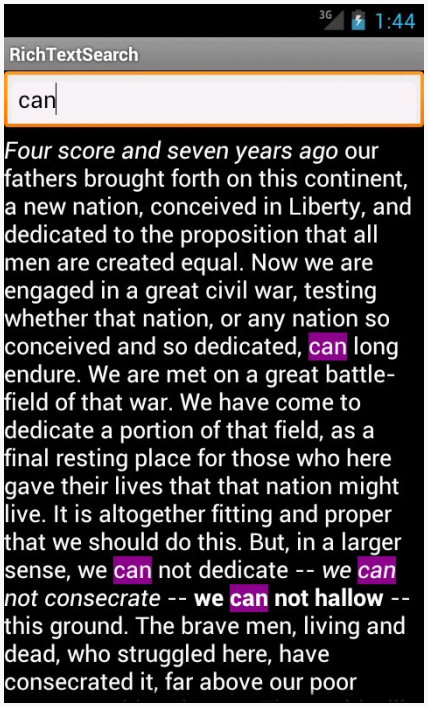


Figure 423: The RichTextSearch sample, after searching on "can"

Saving Rich Text

`SpannableString` and `SpannedString` are not `Serializable`. There is no built-in way to persist them directly.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, `Html.toHtml()` will convert a `Spanned` object into corresponding HTML, for all `CharacterStyle` objects that can be readily converted into HTML. You can then persist the resulting HTML any place you would persist a `String` (e.g., database column).

In principle, you could create other similar conversion code, such as something to take a `Spanned` and return the corresponding Markdown source.

Manipulating Rich Text

The `TextUtils` class has many utility methods that manipulate a `CharSequence`, to allow you to do things that you might ordinarily have done just with methods on `String`. These utility methods will work with any `CharSequence`, including `SpannedString` and `SpannableString`.

Some are specifically aimed at `Spanned` objects, such as `copySpansFrom()` (to apply formatting from one `CharSequence` onto another). Some are clones of `String` equivalents, such as `split()`, `join()`, and `substring()`. Yet others are designed for developers using the `Canvas` 2D drawing API, such as `ellipsize()` and `commaEllipsize()` for intelligently truncating messages.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Animators

Users like things that move. Or fade, spin, or otherwise offer a dynamic experience.

Much of the time, such animations are handled for us by the framework. We do not have to worry about sliding rows in a `ListView` when the user scrolls, or as the user pans around a `ViewPager`, and so forth.

However, sometimes, we will need to add our own animations, where we want effects that either are not provided by the framework innately or are simply different (e.g., want something to slide off the bottom of the screen, rather than off the left edge).

Android had an animation framework back in the beginning, one that is still available for you today. However, Android 3.0 introduced a new *animator* framework that is going to be Android's primary focus for animated effects going forward. Many, but not all, of the animator framework capabilities are available to us as developers via a backport.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Also, you should read the chapter on custom views, to be able to make sense of one of the samples.

ViewPropertyAnimator

Let's say that you want to fade out a widget, instead of simply setting its visibility to `INVISIBLE` or `GONE`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For a widget whose name is v, on API Level 11 or higher, that is as simple as:

```
v.animate().alpha(0);
```

Here, "alpha" refers to the "alpha channel". An alpha of 1 is normal opacity, while an alpha of 0 is completely transparent, with values in between representing various levels of translucence.

That may seem rather simple. The good news is, it *really* is that easy. Of course, there is a lot more you can do here, and you might have to worry about supporting older Android versions, and we need to think about things other than fading widgets in and out, and so forth.

First, though, let's consider what is really going on when we call `animate()` on a widget on API Level 11+.

Native Implementation

The call to `animate()` returns an instance of `ViewPropertyAnimator`. This object allows us to build up a description of an animation to be performed, such as calling `alpha()` to change the alpha channel value. `ViewPropertyAnimator` uses a so-called fluent interface, much like the various builder classes (e.g., `Notification.Builder`) — calling a method on a `ViewPropertyAnimator()` usually returns the `ViewPropertyAnimator` itself. This allows you to build up an animation via a chained series of method calls, starting with that call to `animate()` on the widget.

You will note that we do not end the chain of method calls with something like a `start()` method. `ViewPropertyAnimator` will automatically arrange to start the animation once we return control of the main application thread back to the framework. Hence, we do not have to explicitly start the animation.

You will also notice that we did not indicate any particulars about how the animation should be accomplished, beyond stating the ending alpha channel value of 0. `ViewPropertyAnimator` will use some standard defaults for the animation, such as a default duration, to determine how quickly Android changes the alpha value from its starting point to 0. Most of those particulars can be overridden from their defaults via additional methods called on our `ViewPropertyAnimator`, such as `setDuration()` to provide a duration in milliseconds.

There are four standard animations that `ViewPropertyAnimator` can perform:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. Changes in alpha channel values, for fading widgets in and out
2. Changes in widget position, by altering the X and Y values of the upper-left corner of the widget, from wherever on the screen it used to be to some new value
3. Changes in the widget's rotation, around any of the three axes
4. Changes in the widget's size, where Android can scale the widget by some percentage to expand or shrink it

We will see an example of changing a widget's position, using the `translationXBy()` method, later in this chapter.

You are welcome to use more than one animation effect simultaneously, such as using both `alpha()` and `translationXBy()` to slide a widget horizontally and have it fade in or out.

There are other aspects of the animation that you can control. By default, the animation happens linearly — if we are sliding 500 pixels in 500ms, the widget will move evenly at 1 pixel/ms. However, you can specify a different "interpolator" to override that default linear behavior (e.g., start slow and accelerate as the animation proceeds). You can attach a listener object to find out about when the animation starts and ends. And, you can specify `withLayer()` to indicate that Android should try to more aggressively use hardware acceleration for an animation, a concept that we will get into in greater detail later in this chapter.

To see this in action, take a look at the `Animation/AnimatorFade` sample app.

The app consists of a single activity (`MainActivity`). It uses a layout that is dominated by a single `TextView` widget, whose ID is `fadee`:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TextView
    android:id="@+id/fadee"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:text="@string/fading_out"
    android:textAppearance="?android:attr/textAppearanceLarge"
    tools:context=".MainActivity"/>

</RelativeLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In `onCreate()`, we load up the layout and get our hands on the `fadee` widget:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    fadee=(TextView)findViewById(R.id.fadee);
  }
```

`MainActivity` itself implements `Runnable`, and our `run()` method will perform some animated effects:

```
  @Override
  public void run() {
    if (fadingOut) {
      fadee.animate().alpha(0).setDuration(PERIOD);
      fadee.setText(R.string.fading_out);
    }
    else {
      fadee.animate().alpha(1).setDuration(PERIOD);
      fadee.setText(R.string.coming_back);
    }

    fadingOut=!fadingOut;

    fadee.postDelayed(this, PERIOD);
  }
```

Specifically, if we are to fade out the `TextView` (as we are at the outset, we use `ViewPropertyAnimator` to fade out the widget over a certain period (`fadee.animate().alpha(0).setDuration(PERIOD);`) and set the caption of the `TextView` to a value indicating that we are fading out. If we are to be fading back in, we perform the opposite animation and set the caption to a different value. We then flip the `fadingOut` `boolean` for the next pass and use `postDelayed()` to reschedule ourselves to run after the period has elapsed.

To complete the process, we `run()` our code initially in `onResume()` and cancel the `postDelayed()` loop in `onPause()`:

```
  @Override
  public void onResume() {
    super.onResume();

    run();
  }

  @Override
  public void onPause() {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    fadee.removeCallbacks(this);

    super.onPause();
  }
```

The result is that the `TextView` smoothly fades out and in, alternating captions as it goes.

However, it would be really unpleasant if all this animator goodness worked only on API Level 11+. Fortunately for us, somebody wrote a backport... somebody with whom you are already familiar.

Backport Via NineOldAndroids

Jake Wharton, author of ActionBarSherlock, ViewPagerIndicator, and other libraries, also wrote NineOldAndroids. This is, in effect, a backport of `ViewPropertyAnimator` and its underpinnings. There are some slight changes in how you use it, because NineOldAndroids is simply a library. It cannot add methods to existing classes (like adding `animate()` to `View`), nor can it add capabilities that the underlying firmware simply lacks. But, it may cover many of your animator needs, even if the name is somewhat inexplicable, and it works going all the way back to API Level 1, ensuring that it will cover any Android release that you care about.

As with ActionBarSherlock, NineOldAndroids is an Android library project.

Eclipse users will need to download that project (look in the `library/` directory of the ZIP archive) and import the project. Android Studio users can add a `compile` statement to their `dependencies` closure in `build.gradle` to pull in `com.nineoldandroids:library:...` (for some version indicated by `...`) from `mavenCentral()`.

Since NineOldAndroids cannot add `animate()` to `View`, the recommended approach is to use a somewhat obscure feature of Java: imported static methods. An `import static` statement, referencing a particular static method of a class, makes that method available as if it were a static method on the class that you are writing, or as some sort of global function. NineOldAndroids has an `animate()` method that you can import this way, so instead of `v.animate()`, you use `animate(v)` to accomplish the same end. Everything else is the same, except perhaps some imports, to reference NineOldAndroids instead of the native classes.

You can see this in the `Animation/AnimatorFadeBC` sample app.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In addition to having the NineOldAndroids JAR in `libs/`, the only difference between this edition and the previous sample is in how the animation is set up. Instead of lines like:

```
fadee.animate().alpha(0).setDuration(PERIOD);
```

we have:

```
animate(fadee).alpha(0).setDuration(PERIOD);
```

This takes advantage of our static import:

```
import static com.nineoldandroids.view.ViewPropertyAnimator.animate;
```

If the static import makes you queasy, you are welcome to simply import the `com.nineoldandroids.view.ViewPropertyAnimator` class, rather than the static method, and call the `animate()` method on `ViewPropertyAnimator`:

```
ViewPropertyAnimator.animate(fadee).alpha(0).setDuration(PERIOD);
```

The Foundation: Value and Object Animators

`ViewPropertyAnimator` itself is a layer atop of a more primitive set of animators, known as value and object animators.

A `ValueAnimator` handles the core logic of transitioning some value, from an old to a new value, over a period of time. `ValueAnimator` offers replaceable "interpolators", which will determine how the values change from start to finish over the animation period (e.g., start slowly, accelerate, then end slowly). `ValueAnimator` also handles the concept of a "repeat mode", to indicate if the animation should simply happen once, a fixed number of times, or should infinitely repeat (and, in the latter cases, whether it does so always transitioning from start to finish or if it reverses direction on alternate passes, going from finish back to start).

What `ValueAnimator` does *not* do is actually change anything. It is merely computing the different values based on time. You can call `getAnimatedValue()` to find out the value at any point in time, or you can call `addUpdateListener()` to register a listener object that will be notified of each change in the value, so that change can be applied somewhere.

Hence, what tends to be a bit more popular is `ObjectAnimator`, a subclass of `ValueAnimator` that automatically applies the new values. `ObjectAnimator` does this

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

by calling a setter method on some object, where you supply the object and the "property name" used to derive the getter and setter method names. For example, if you request a property name of `foo`, `ObjectAnimator` will try to call `getFoo()` and `setFoo()` methods on your supplied object.

As with `ViewPropertyAnimator`, `ValueAnimator` and `ObjectAnimator` are implemented natively in API Level 11 and are available via the NineOldAndroids backport as well.

To see what `ObjectAnimator` looks like in practice, let us examine the `Animation/ObjectAnimator` sample app.

Once again, our activity's layout is pretty much just a centered `TextView`, here named `word`:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TextView
    android:id="@+id/word"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:textAppearance="?android:attr/textAppearanceLarge"
    tools:context=".MainActivity"/>

</RelativeLayout>
```

The objective of our activity is to iterate through 25 words, showing one at a time in the `TextView`:

```
package com.commonsware.android.animator.obj;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import com.nineoldandroids.animation.ObjectAnimator;
import com.nineoldandroids.animation.ValueAnimator;

public class MainActivity extends Activity {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private TextView word=null;
  int position=0;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    word=(TextView)findViewById(R.id.word);

    ValueAnimator positionAnim = ObjectAnimator.ofInt(this, "wordPosition", 0,
24);
    positionAnim.setDuration(12500);
    positionAnim.setRepeatCount(ValueAnimator.INFINITE);
    positionAnim.setRepeatMode(ValueAnimator.RESTART);
    positionAnim.start();
  }

  public void setWordPosition(int position) {
    this.position=position;
    word.setText(items[position]);
  }

  public int getWordPosition() {
    return(position);
  }
}
```

To accomplish this, we use NineOldAndroids version of `ObjectAnimator`, saying that we wish to "animate" the `wordPosition` property of the activity itself, from 0 to 24. We configure the animation to run for 12.5 seconds (i.e., 500ms per word) and to repeat indefinitely by restarting the animation from the beginning on each pass. We then call `start()` to kick off the animation.

For this to work, though, we need `getWordPosition()` and `setWordPosition()` accessor methods for the theoretical `wordPosition` property. In our case, the "word position" is simply an integer data member of the activity, which we return in `getWordPosition()` and update in `setWordPosition()`. However, we also update the `TextView` in `setWordPosition()`, to display the word at that position.

The net effect is that words appear in our `TextView`, changing on average every 500ms.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Animating Custom Types

In the previous section, we animated an `int` property of an `Activity`. That works, because Android knows how to compute `int` values between the start and end position, through simple math.

But, what if we wanted to animate something that is not a simple number? For example, what if we want to animate a `Color`, or a `LatLng` from Maps V2, or a `TastyTreat` class of our own design?

So long as *we* can perform the calculations, we can animate a type of anything we want, using `TypeEvaluator` and `ofObject()` on `ObjectAnimator`.

A `TypeEvaluator` is a simple interface, containing a single method that we need to override: `evaluate()`. However, `TypeEvaluator` uses generics, and so our implementation will actually be of some concrete class (e.g., a `TypeEvaluator` of `TastyTreat`). Our job in `evaluate()` is to return a value of our designated type (e.g., `TastyTreat`) given three inputs:

1. The initial value for our animation range, in the form of our designated type
2. The end value for our animation range, in the form of our designated type
3. The fraction along that range that represents how much we have moved from the initial value to the end value

Note that the fraction is not limited to being between 0 and 1, as certain interpolators (e.g., an overshoot interpolator) might result in a fraction being negative (e.g., we overshot past the initial value) or greater than one (e.g., we overshot past the end value).

For example, to have a `TypeEvaluator` of `Color`, we might have `evaluate()` generate a new `Color` instance based upon applying the fraction to the initial and end red, green, blue, and alpha channels.

To use a `TypeEvaluator`, instead of `ofInt()`, `ofFloat()`, or similar simple factory methods on `ObjectAnimator`, we use `ofObject()`. `ofObject()` takes the object to be animated, the property to be animated, the `TypeEvaluator` to assist in the actual animation, and the final value of the animation (or, optionally, a series of waypoints to be animated along).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A flavor of `ofObject()` that takes the property name — akin to the `wordPosition` `ofInt()` used in the previous section — has been around since API Level 11. API Level 14 added an `ofObject()` method that takes a `Property` value instead of the name of the property. This version has the added benefit of type-safety, as it can ensure that your object to be animated, `TypeEvaluator`, and final position are all of the same type.

You can see an example of using `TypeEvaluator` this way in the chapter on Maps V2, as we animate the movement of a map marker from a starting point to an ending point.

Hardware Acceleration

Animated effects operate much more smoothly with hardware acceleration. There are two facets to employing hardware acceleration for animations: enabling it overall and directing its use for the animations themselves.

Hardware acceleration is enabled overall on Android devices running Android 4.0 or higher (API Level 14). On Android 3.x, hardware acceleration is available but is disabled by default — use `android:hardwareAccelerated="true"` in your `<application>` or `<activity>` element in the manifest to enable it on those versions. Hardware acceleration for 2D graphics operations like widget animations is not available on older versions of Android.

While this will provide some benefit across the board, you may also wish to consider rendering animated widgets or containers in an off-screen buffer, or "hardware layer", that then gets applied to the screen via the GPU. In particular, the GPU can apply certain animated transformations to a hardware layer without forcing software to redraw the widgets or containers (e.g., what happens when you `invalidate()` them). As it turns out, these GPU-enhanced transformations match the ones supported by `ViewPropertyAnimator`:

1. Changes in alpha channel values, for fading widgets in and out
2. Changes in widget position, by altering the X and Y values of the upper-left corner of the widget, from wherever on the screen it used to be to some new value
3. Changes in the widget's rotation, around any of the three axes
4. Changes in the widget's size, where Android can scale the widget by some percentage to expand or shrink it

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

By having the widget be rendered in a hardware layer, these `ViewPropertyAnimator` operations are significantly more efficient than before.

However, since hardware layers take up video memory, generally you do not want to keep a widget or container in a hardware layer indefinitely. Instead, the recommended approach is to have the widget or container be rendered in a hardware layer only while the animation is ongoing, by calling `setLayerType()` for `LAYER_TYPE_HARDWARE` before the animation begins, then calling `setLayerType()` for `LAYER_TYPE_NONE` (i.e., return to default behavior) when the animation completes. Or, for `ViewPropertyAnimator` on API Level 16 and higher, use `withLayer()` in the fluent interface to have it apply the hardware layer automatically just for the animation duration.

We will see examples of using hardware acceleration this way in the next section.

The Three-Fragment Problem

If you have used an Android tablet, there is a decent chance that you have used the Gmail app on that tablet. Gmail organizes its landscape main activity into two panes, one on the left taking up ~30% of the screen, and one on the right taking up the remainder:

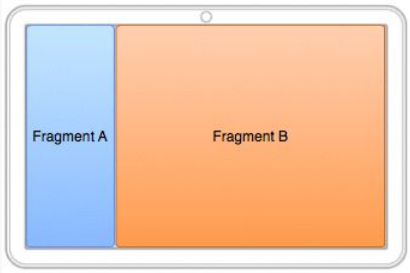


Figure 424: Gmail Fragments (image courtesy of Google and AOSP)

Gmail has a very specific navigation mode in its main activity when viewed in landscape on a tablet, where upon some UI event (e.g., tapping on something in the right-hand area):

- The original left-hand fragment (Fragment A) slides off the screen to the left
- The original right-hand fragment (Fragment B) slides to the left edge of the screen and shrinks to take up the spot vacated by Fragment A

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Another fragment (Fragment C) slides in from the right side of the screen and takes up the spot vacated by Fragment B

And a BACK button press reverses this operation.

This is a bit tricky to set up, leading to the author of this book posting a question on Stack Overflow to get input. Here, we will examine one of the results of that discussion, based in large part on the implementation of the AOSP Email app, which has a similar navigation flow. The other answers on that question may have merit in other scenarios as well.

You can see one approach for implementing the three-pane solution in the `Animation/ThreePane` sample app.

The ThreePaneLayout

The logic to handle the animated effects is encapsulated in a `ThreePaneLayout` class. It is designed to be used in a layout XML resource where you supply the contents of the three panes, sizing the first two as you want, with the third "pane" having zero width at the outset:

```
<com.commonsware.android.anim.threepane.ThreePaneLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/root"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <FrameLayout
    android:id="@+id/left"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="3"/>

  <FrameLayout
    android:id="@+id/middle"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="7"/>

  <Button
    android:layout_width="0dp"
    android:layout_height="match_parent"/>

</com.commonsware.android.anim.threepane.ThreePaneLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

ThreePaneLayout itself is a subclass of LinearLayout, set up to always be horizontal, regardless of what might be set in the layout XML resource.

```
public ThreePaneLayout(Context context, AttributeSet attrs) {
  super(context, attrs);
  initSelf();
}

void initSelf() {
  setOrientation(HORIZONTAL);
}
```

When the layout finishes inflating, we grab the three panes (defined as the first three children of the container) and stash them in data members named left, middle, and right, with matching getter methods:

```
@Override
public void onFinishInflate() {
  super.onFinishInflate();

  left=getChildAt(0);
  middle=getChildAt(1);
  right=getChildAt(2);
}

public View getLeftView() {
  return(left);
}

public View getMiddleView() {
  return(middle);
}

public View getRightView() {
  return(right);
}
```

The major operational API, from the standpoint of an activity using ThreePaneLayout, is hideLeft() and showLeft(). hideLeft() will switch from showing the left and middle widgets in their original size and position to showing the middle and right widgets wherever left and middle had been originally. showLeft() reverses the operation.

The problem is that, initially, we do not know where the widgets are or how big they are, as that should be able to be set from the layout XML resource and are not known until the ThreePaneLayout is actually applied to the screen. Hence, we lazy-retrieve those values in hideLeft(), plus remove any weights that had been originally defined, setting the actual pixel widths on the widgets instead:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public void hideLeft() {
  if (leftWidth == -1) {
    leftWidth=left.getWidth();
    middleWidthNormal=middle.getWidth();
    resetWidget(left, leftWidth);
    resetWidget(middle, middleWidthNormal);
    resetWidget(right, middleWidthNormal);
    requestLayout();
  }

  translateWidgets(-1 * leftWidth, left, middle, right);

  ObjectAnimator.ofInt(this, "middleWidth", middleWidthNormal,
                       leftWidth).setDuration(ANIM_DURATION).start();
}
```

The work to change the weights into widths is handled in `resetWidget()`:

```
private void resetWidget(View v, int width) {
  LinearLayout.LayoutParams p=
      (LinearLayout.LayoutParams)v.getLayoutParams();

  p.width=width;
  p.weight=0;
}
```

After the lazy-initialization and widget cleanup, we perform the two animations. `translateWidgets()` will slide each of our three widgets to the left by the width of the `left` widget, using a `ViewPropertyAnimator` and a hardware layer:

```
private void translateWidgets(int deltaX, View... views) {
  for (final View v : views) {
    v.setLayerType(View.LAYER_TYPE_HARDWARE, null);

    v.animate().translationXBy(deltaX).setDuration(ANIM_DURATION)
     .setListener(new AnimatorListenerAdapter() {
       @Override
       public void onAnimationEnd(Animator animation) {
         v.setLayerType(View.LAYER_TYPE_NONE, null);
       }
     });
  }
}
```

The resize animation — to set the `middle` size to be what `left` had been – is handled via an `ObjectAnimator`, for a theoretical property of `middleWidth` on `ThreePaneLayout`. That is backed by a `setMiddleWidth()` method that adjusts the `width` property of the `middle` widget's `LayoutParams` and triggers a redraw:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @SuppressWarnings("unused")
  private void setMiddleWidth(int value) {
    middle.getLayoutParams().width=value;
    requestLayout();
  }
```

The `showLeft()` method simply performs those two animations in reverse:

```
  public void showLeft() {
    translateWidgets(leftWidth, left, middle, right);

    ObjectAnimator.ofInt(this, "middleWidth", leftWidth,
                         middleWidthNormal).setDuration(ANIM_DURATION)
                  .start();
  }
```

Using the ThreePaneLayout

The sample app uses one activity (`MainActivity`) and one fragment (`SimpleListFragment`) to set up and use the `ThreePaneLayout`. The objective is a UI that roughly mirrors that of Gmail and the AOSP Email app: a list on the left, a list in the middle (whose contents are based on the item chosen in the left list), and something else on the right (whose contents are based on the item chosen in the middle list).

`SimpleListFragment` is used for both lists. Its `newInstance()` factory method is handed the list of strings to display. `SimpleListFragment` just loads those into its `ListView`, also setting up `CHOICE_MODE_SINGLE` for use with the `activated` style, and routing all clicks on the list to the `MainActivity` that hosts the fragment:

```
package com.commonsware.android.anim.threepane;

import android.app.ListFragment;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import java.util.ArrayList;
import java.util.Arrays;

public class SimpleListFragment extends ListFragment {
  private static final String KEY_CONTENTS="contents";

  public static SimpleListFragment newInstance(String[] contents) {
    return(newInstance(new ArrayList<String>(Arrays.asList(contents))));
  }

  public static SimpleListFragment newInstance(ArrayList<String> contents) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    SimpleListFragment result=new SimpleListFragment();
    Bundle args=new Bundle();

    args.putStringArrayList(KEY_CONTENTS, contents);
    result.setArguments(args);

    return(result);
  }

  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);

    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
    setContents(getArguments().getStringArrayList(KEY_CONTENTS));
  }

  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    ((MainActivity)getActivity()).onListItemClick(this, position);
  }

  void setContents(ArrayList<String> contents) {
    setListAdapter(new ArrayAdapter<String>(
                                          getActivity(),
                                          R.layout.simple_list_item_1,
                                          contents));
  }
}
```

`MainActivity` populates the `left` `FrameLayout` with a `SimpleListFragment` in `onCreate()`, if the fragment does not already exist (e.g., from a configuration change). When an item in the left list is clicked, `MainActivity` populates the `middle` `FrameLayout`. When an item in the middle list is clicked, it sets the caption of the `right` `Button` and uses `hideLeft()` to animate that `Button` onto the screen, hiding the `left` list. If the user presses BACK, and our `left` list is not showing, `MainActivity` calls `showLeft()` to reverse the animation:

```
package com.commonsware.android.anim.threepane;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Button;
import java.util.ArrayList;

public class MainActivity extends Activity {
  private static final String KEY_MIDDLE_CONTENTS="middleContents";
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    "pellentesque", "augue", "purus" };
private boolean isLeftShowing=true;
private SimpleListFragment middleFragment=null;
private ArrayList<String> middleContents=null;
private ThreePaneLayout root=null;

@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  root=(ThreePaneLayout)findViewById(R.id.root);

  if (getFragmentManager().findFragmentById(R.id.left) == null) {
    getFragmentManager().beginTransaction()
                        .add(R.id.left,
                             SimpleListFragment.newInstance(items))
                        .commit();
  }

  middleFragment=
      (SimpleListFragment)getFragmentManager().findFragmentById(R.id.middle);
}

@Override
public void onBackPressed() {
  if (!isLeftShowing) {
    root.showLeft();
    isLeftShowing=true;
  }
  else {
    super.onBackPressed();
  }
}

@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  outState.putStringArrayList(KEY_MIDDLE_CONTENTS, middleContents);
}

@Override
protected void onRestoreInstanceState(Bundle inState) {
  middleContents=inState.getStringArrayList(KEY_MIDDLE_CONTENTS);
}

void onListItemClick(SimpleListFragment fragment, int position) {
  if (fragment == middleFragment) {
    ((Button)root.getRightView()).setText(middleContents.get(position));

    if (isLeftShowing) {
      root.hideLeft();
      isLeftShowing=false;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      }
    }
    else {
      middleContents=new ArrayList<String>();

      for (int i=0; i < 20; i++) {
        middleContents.add(items[position] + " #" + i);
      }

      if (getFragmentManager().findFragmentById(R.id.middle) == null) {
        middleFragment=SimpleListFragment.newInstance(middleContents);
        getFragmentManager().beginTransaction()
                            .add(R.id.middle, middleFragment).commit();
      }
      else {
        middleFragment.setContents(middleContents);
      }
    }
  }
}
```

The Results

If you run this app on a landscape tablet running API Level 11 or higher, you start off with a single list of words on the left:

Figure 425: ThreePane, As Initially Launched

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Clicking on a word brings up a second list, taking up the rest of the screen, with numbered entries based upon the clicked-upon word:

Figure 426: ThreePane, After Clicking a Word

Clicking on an entry in the second list starts the animation, sliding the first list off to the left, sliding the second list into the space vacated by the first list, and sliding in a "detail view" into the right portion of the screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

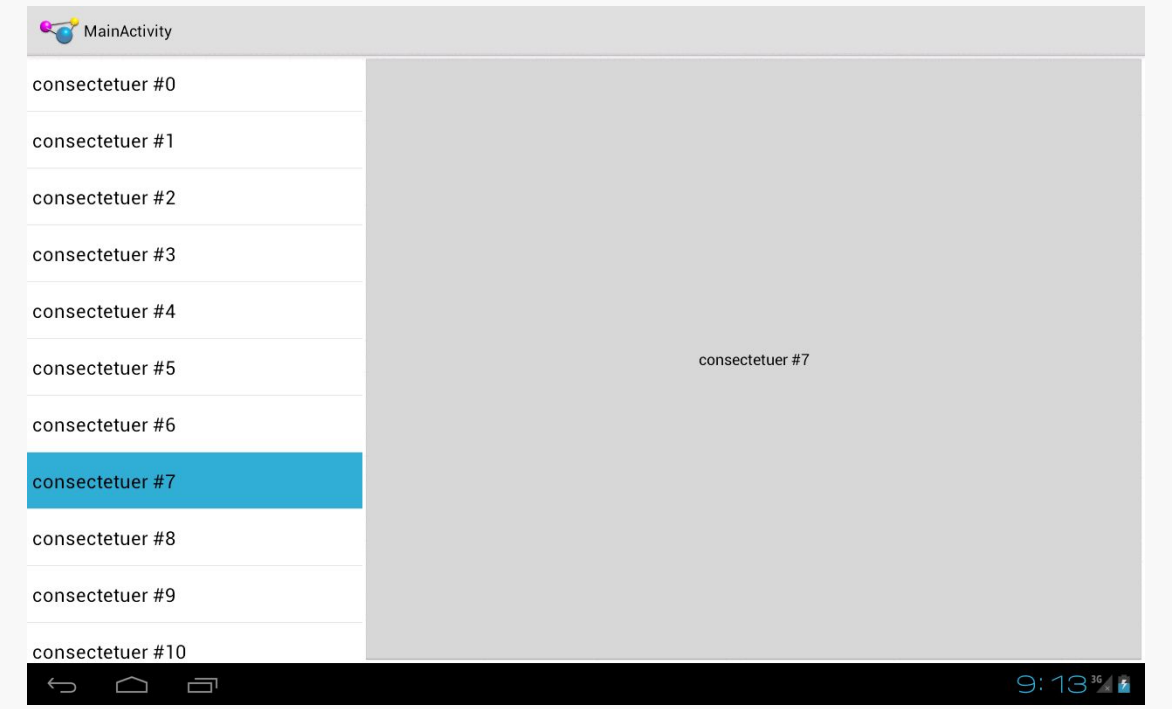


Figure 427: ThreePane, After Clicking a Numbered Word

Pressing BACK once will reverse the animation, restoring you to the two-list perspective.

The Backport

The `ThreePane` sample described above uses the native API Level 11 version of the animator framework and the native implementation of fragments. However, the same approach can work using the Android Support package's version of fragments and NineOldAndroids. You can see this in the `Animation/ThreePaneBC` sample app.

Besides changing the import statements and adding the NineOldAndroids JAR file, the only other changes of substance were:

- Using `ViewPropertyAnimator.animate(v)` instead of `v.animate()` in `translateWidgets()`
- Conditionally setting the hardware acceleration layers via `setLayerType()` in `translateWidgets()` based upon API level, as that method was only added in API Level 11

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The smoothness of animations, though, will vary by hardware capabilities. For example, on a first-generation Kindle Fire, running Android 2.3, the backport works but is not especially smooth, while the animations are very smooth on more modern hardware where hardware acceleration can be applied.

The Problems

As we will see in the chapter on "jank", there is some stutter in the rendering of this app. Fixing it requires removing the animated change in the width of the middle pane, which in turn makes the animation itself look worse. More details on the analysis can be found in the "jank" chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Legacy Animations

Before `ViewPropertyAnimator` and the rest of the animator framework were added in API Level 11, we had the original `Animation` base class and specialized animations based upon it, like `TranslateAnimation` for movement and `AlphaAnimation` for fades. On the whole, you will want to try to use the animator framework where possible, as the new system is more powerful and efficient than the legacy `Animation` approach. However, particularly for apps where the NineOldAndroids backport is insufficient, you may wish to use the legacy framework.

After an overview of the role of the animation framework, we go in-depth to animate the movement of a widget across the screen. We then look at alpha animations, for fading widgets in and out. We then see how you can get control during the lifecycle of an animation, how to control the acceleration of animations, and how to group animations together for parallel execution. Finally, we see how the same framework can now be used to control the animation for the switching of activities.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the ones on basic resources and basic widgets. Also, you should read the chapter on custom views.

It's Not Just For Toons Anymore

Android has a package of classes (`android.view.animation`) dedicated to animating the movement and behavior of widgets.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

They center around an `Animation` base class that describes what is to be done. Built-in animations exist to move a widget (`TranslateAnimation`), change the transparency of a widget (`AlphaAnimation`), revolve a widget (`RotateAnimation`), and resize a widget (`ScaleAnimation`). There is even a way to aggregate animations together into a composite `Animation` called an `AnimationSet`. Later sections in this chapter will examine the use of several of these animations.

Given that you have an animation, to apply it, you have two main options:

1. You may be using a container that supports animating its contents, such as a `ViewFlipper` or `TextSwitcher`. These are typically subclasses of `ViewAnimator` and let you define the "in" and "out" animations to apply. For example, with a `ViewFlipper`, you can specify how it flips between `Views` in terms of what animation is used to animate "out" the currently-visible `View` and what animation is used to animate "in" the replacement `View`.
2. You can simply tell any `View` to `startAnimation()`, given the `Animation` to apply to itself. This is the technique we will be seeing used in the examples in this chapter.

A Quirky Translation

Animation takes some getting used to. Frequently, it takes a fair bit of experimentation to get it all working as you wish. This is particularly true of `TranslateAnimation`, as not everything about it is intuitive, even to authors of Android books.

Mechanics of Translation

The simple constructor for `TranslateAnimation` takes four parameters describing how the widget should move: the before and after X offsets from the current position, and the before and after Y offsets from the current position. The Android documentation refers to these as `fromXDelta`, `toXDelta`, `fromYDelta`, and `toYDelta`.

In Android's pixel-space, an `(X,Y)` coordinate of `(0,0)` represents the upper-left corner of the screen. Hence, if `toXDelta` is greater than `fromXDelta`, the widget will move to the right, if `toYDelta` is greater than `fromYDelta`, the widget will move down, and so on.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Imagining a Sliding Panel

Some Android applications employ a sliding panel, one that is off-screen most of the time but can be called up by the user (e.g., via a menu) when desired. When anchored at the bottom of the screen, you get a container that slides up from the bottom and slides down and out when being removed.

One way to implement such a panel is to have a container (e.g., a `LinearLayout`) whose contents are absent (`INVISIBLE`) when the panel is closed and is present (`VISIBLE`) when the drawer is open. If we simply toggled `setVisibility()` using the aforementioned values, though, the panel would wink open and closed immediately, without any sort of animation. So, instead, we want to:

1. Make the panel visible and animate it up from the bottom of the screen when we open the panel
2. Animate it down to the bottom of the screen and make the panel invisible when we close the panel

The Aftermath

This brings up a key point with respect to `TranslateAnimation`: the animation temporarily moves the widget, but if you want the widget to stay where it is when the animation is over, you have to handle that yourself. Otherwise, the widget will snap back to its original position when the animation completes.

In the case of the panel opening, we handle that via the transition from `INVISIBLE` to `VISIBLE`. Technically speaking, the panel is always "open", in that we are not, in the end, changing its position. But when the body of the panel is `INVISIBLE`, it takes up no space on the screen; when we make it `VISIBLE`, it takes up whatever space it is supposed to.

Later in this chapter, we will cover how to use animation listeners to accomplish this end for closing the panel.

Introducing SlidingPanel

With all that said, turn your attention to the `Animation/SlidingPanel` sample project and, in particular, the `SlidingPanel` class.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This class implements a layout that works as a panel, anchored to the bottom of the screen. A `toggle()` method can be called by the activity to hide or show the panel. The panel itself is a `LinearLayout`, so you can put whatever contents you want in there.

We use two flavors of `TranslateAnimation`, one for opening the panel and one for closing it.

Here is the opening animation:

```
      anim=new TranslateAnimation(0.0f, 0.0f,
                                  getHeight(),
                                  0.0f);
```

Our `fromXDelta` and `toXDelta` are both `0`, since we are not shifting the panel's position along the horizontal axis. Our `fromYDelta` is the panel's height according to its layout parameters (representing how big we want the panel to be), because we want the panel to start the animation at the bottom of the screen; our `toYDelta` is `0` because we want the panel to be at its "natural" open position at the end of the animation.

Conversely, here is the closing animation:

```
      anim=new TranslateAnimation(0.0f, 0.0f, 0.0f,
                                  getHeight());
```

It has the same basic structure, except the Y values are reversed, since we want the panel to start open and animate to a closed position.

The result is a container that can be closed:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

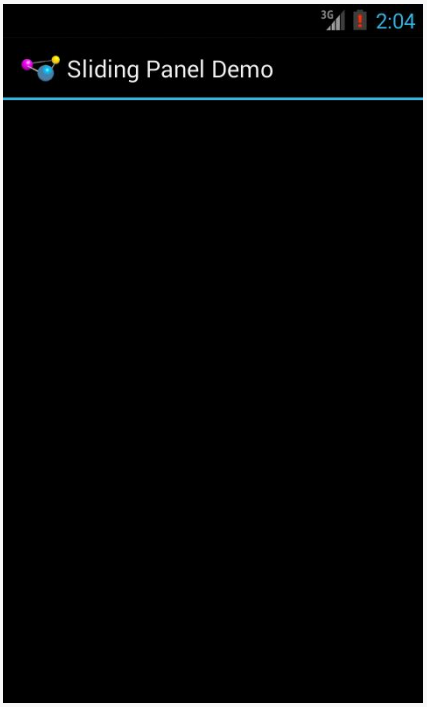


Figure 428: The SlidingPanel sample application, with the panel closed

... or open, in this case toggled via a menu choice in the `SlidingPanelDemo` activity:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

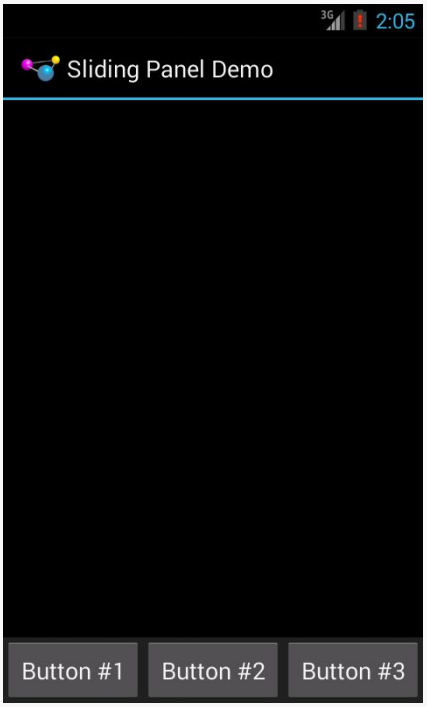


Figure 429: The SlidingPanel sample application, with the panel open

Using the Animation

When setting up an animation, you also need to indicate how long the animation should take. This is done by calling `setDuration()` on the animation, providing the desired length of time in milliseconds.

When we are ready with the animation, we simply call `startAnimation()` on the `SlidingPanel` itself, causing it to move as specified by the `TranslateAnimation` instance.

Fading To Black. Or Some Other Color.

`AlphaAnimation` allows you to fade a widget in or out by making it less or more transparent. The greater the transparency, the more the widget appears to be "fading".

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Alpha Numbers

You may be used to alpha channels, when used in `#AARRGGBB` color notation, or perhaps when working with alpha-capable image formats like PNG.

Similarly, `AlphaAnimation` allows you to change the alpha channel for an entire widget, from fully-solid to fully-transparent.

In Android, a float value of `1.0` indicates a fully-solid widget, while a value of `0.0` indicates a fully-transparent widget. Values in between, of course, represent various amounts of transparency.

Hence, it is common for an `AlphaAnimation` to either start at `1.0` and smoothly change the alpha to `0.0` (a fade) or vice versa.

Animations in XML

With `TranslateAnimation`, we showed how to construct the animation in Java source code. One can also create animation resources, which define the animations using XML. This is similar to the process for defining layouts, albeit much simpler.

For example, there is a second animation project, [Animation/SlidingPanelEx](), which demonstrates a panel that fades out as it is closed. In there, you will find a `res/anim/` directory, which is where animation resources should reside. In there, you will find `fade.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
  android:fromAlpha="1.0"
  android:toAlpha="0.0" />
```

The name of the root element indicates the type of animation (in this case, alpha for an `AlphaAnimation`). The attributes specify the characteristics of the animation, in this case a fade from `1.0` to `0.0` on the alpha channel.

This XML is the same as calling `new AlphaAnimation(1.0f,0.0f)` in Java.

Using XML Animations

To make use of XML-defined animations, you need to inflate them, much as you might inflate a `View` or `Menu` resource. This is accomplished by using the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

loadAnimation() static method on the AnimationUtils class, seen here in our SlidingPanel constructor:

```
public SlidingPanel(final Context ctxt, AttributeSet attrs) {
  super(ctxt, attrs);

  TypedArray a=ctxt.obtainStyledAttributes(attrs,
                                           R.styleable.SlidingPanel,
                                           0, 0);

  speed=a.getInt(R.styleable.SlidingPanel_speed, 300);

  a.recycle();

  fadeOut=AnimationUtils.loadAnimation(ctxt, R.anim.fade);
}
```

Here, we are loading our fade animation, given a Context. This is being put into an Animation variable, so we neither know nor care that this particular XML that we are loading defines an AlphaAnimation instead of, say, a RotateAnimation.

When It's All Said And Done

Sometimes, you need to take action when an animation completes.

For example, when we close the panel, we want to use a TranslationAnimation to slide it down from the open position to closed... then *keep* it closed. With the system used in SlidingPanel, keeping the panel closed is a matter of calling setVisibility() on the contents with INVISIBLE.

However, you cannot do that when the animation begins; otherwise, the panel is gone by the time you try to animate its motion.

Instead, you need to arrange to have it become invisible when the animation ends. To do that, you use an animation listener.

An animation listener is simply an instance of the AnimationListener interface, provided to an animation via setAnimationListener(). The listener will be invoked when the animation starts, ends, or repeats (the latter courtesy of CycleInterpolator, discussed later in this chapter). You can put logic in the onAnimationEnd() callback in the listener to take action when the animation finishes.

For example, here is the AnimationListener for SlidingPanel:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  Animation.AnimationListener collapseListener=new Animation.AnimationListener()
{
    public void onAnimationEnd(Animation animation) {
      setVisibility(View.INVISIBLE);
    }

    public void onAnimationRepeat(Animation animation) {
      // not needed
    }

    public void onAnimationStart(Animation animation) {
      // not needed
    }
  };
```

All we do is set our content's visibility to be `INVISIBLE`, thereby closing the panel.

Loose Fill

You will see attributes, available on `Animation`, named `android:fillEnabled` and `android:fillAfter`. Reading those, you may think that you can dispense with the `AnimationListener` and just use those to arrange to have your widget wind up being "permanently" in the state represented by the end of the animation. All you would have to do is set each of those to true in your animation XML (or the equivalent in Java), and you would be set.

At least for `TranslateAnimation`, you would be mistaken.

It actually will look like it works — the animated widgets will be drawn in their new location. However, if those widgets are clickable, they will not be clicked in their new location, but rather in their old one. This, of course, is not terribly useful.

Hence, even though it is annoying, you will want to use the `AnimationListener` techniques described in this chapter.

Hit The Accelerator

In addition to the `Animation` classes themselves, Android also provides a set of `Interpolator` classes. These provide instructions for how an animation is supposed to behave during its operating period.

For example, the `AccelerateInterpolator` indicates that, during the duration of an animation, the rate of change of the animation should begin slowly and accelerate

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

until the end. When applied to a `TranslateAnimation`, for example, the sliding movement will start out slowly and pick up speed until the movement is complete.

There are several implementations of the `Interpolator` interface besides `AccelerateInterpolator`, including:

1. `AccelerateDecelerateInterpolator`, which starts slowly, picks up speed in the middle, and slows down again at the end
2. `DecelerateInterpolator`, which starts quickly and slows down towards the end
3. `LinearInterpolator`, the default, which indicates the animation should proceed smoothly from start to finish
4. `CycleInterpolator`, which repeats an animation for a number of cycles, following the `AccelerateDecelerateInterpolator` pattern (slow, then fast, then slow)

To apply an interpolator to an animation, simply call `setInterpolator()` on the animation with the `Interpolator` instance, such as the following line from `SlidingPanel`:

```
    anim.setInterpolator(new AccelerateInterpolator(1.0f));
```

You can also specify one of the stock interpolators via the `android:interpolator` attribute in your animation XML file.

Animate. Set. Match.

For the `Animation/SlidingPanelEx` project, though, we want the panel to slide open, but also fade when it slides closed. This implies two animations working at the same time (a fade and a slide). Android supports this via the `AnimationSet` class.

An `AnimationSet` is itself an `Animation` implementation. Following the composite design pattern, it simply cascades the major `Animation` events to each of the animations in the set.

To create a set, just create an `AnimationSet` instance, add the animations, and configure the set. For example, here is the logic from the `SlidingPanel` implementation in `Animation/SlidingPanelEx`:

```
  public void toggle() {
    TranslateAnimation anim=null;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    AnimationSet set=new AnimationSet(true);

    isOpen=!isOpen;

    if (isOpen) {
      setVisibility(View.VISIBLE);
      anim=new TranslateAnimation(0.0f, 0.0f,
                                  getHeight(),
                                  0.0f);
    }
    else {
      anim=new TranslateAnimation(0.0f, 0.0f, 0.0f,
                                  getHeight());
      anim.setAnimationListener(collapseListener);
      set.addAnimation(fadeOut);
    }

    set.addAnimation(anim);
    set.setDuration(speed);
    set.setInterpolator(new AccelerateInterpolator(1.0f));
    startAnimation(set);
  }
```

If the panel is to be opened, we make the contents visible (so we can animate the motion upwards), and create a `TranslateAnimation` for the upward movement. If the panel is to be closed, we create a `TranslateAnimation` for the downward movement, but also add a pre-defined `AlphaAnimation` (`fadeOut`) to an `AnimationSet`. In either case, we add the `TranslateAnimation` to the set, give the set a duration and interpolator, and run the animation.

Active Animations

Starting with Android 1.5, users could indicate if they wanted to have inter-activity animations: a slide-in/slide-out effect as they switched from activity to activity. However, at that time, they could merely toggle this setting on or off, and applications had no control over these animations whatsoever.

Starting in Android 2.0, though, developers have a bit more control. Specifically:

1. Developers can call `overridePendingTransition()` on an `Activity`, typically after calling `startActivity()` to launch another activity or `finish()` to close up the current activity. The `overridePendingTransition()` indicates an in/out animation pair that should be applied as control passes from this activity to the next one, whether that one is being started (`startActivity()`) or is the one previous on the stack (`finish()`).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. Developers can start an activity via an `Intent` containing the `FLAG_ACTIVITY_NO_ANIMATION` flag. As the name suggests, this flag requests that animations on the transitions involving this activity be suppressed.

These are prioritized as follows:

- Any call to `overridePendingTransition()` is always taken into account
- Lacking that, `FLAG_ACTIVITY_NO_ANIMATION` will be taken into account
- In the normal case, where neither of the two are used, whatever the user's preference, via the Settings application, is applied

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Custom Drawables

Many times, our artwork can simply be some PNG or JPEG files, perhaps with different variations in different resource directories by density.

Sometimes, though, we need something more.

In addition to supporting standard PNG and JPEG files, Android has a number of custom drawable resource formats — mostly written in XML — that handle specific scenarios.

For example, you may wish to customize "the background" of a `Button`, but a `Button` really has several different background images for different circumstances (normal, pressed, focused, disabled, etc.). Android has a certain type of drawable resource that aggregates other drawable resources, indicating which of those other resources should be used in different circumstances (e.g., for a normal button use X, for a disabled button use Y).

In this chapter, we will explore these non-traditional types of "drawables" and how you can use them within your apps.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the ones on basic resources and basic widgets.

Having read the chapters on animators and legacy animations would be useful.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

ColorDrawable

The simplest XML drawable format, by far, is for `ColorDrawable`. Not surprisingly, this defines a `Drawable` that is a solid color.

So, you can have a `res/drawable/thing.xml` file, containing something like this:

```
<color xmlns:android="http://schemas.android.com/apk/res/android"
    android:color="#80FF00FF"/>
```

From there, you can use `@drawable/thing` or `R.drawable.thing` in the same places that you would use any other drawable resource.

AnimationDrawable

The original way of doing animation on the Web was via the animated GIF. An individual GIF file could contain many frames, and the browser would switch between those frames to display a basic animated effect. This was used by Web designers for things both good (animated progress "spinners") and bad ("hit the monkey" ad banners).

Android, on the whole, does not support animated GIF files, certainly not as regular images for use with widgets like `ImageView`.

However, there are times where having this sort of frame-by-frame animation would be useful. For example, in another chapter, we will look at `ProgressBar`, which is Android's primary way of demonstrating progress of background work. You may wish to customize the "spinning wheel" image that Android uses by default, to match your app's color scheme, or to spin your company logo, or whatever. On the Web, particularly on older browsers, you might use an animated GIF for that — on Android, that is not an option.

An `AnimationDrawable`, though, *is* an option.

`AnimationDrawable` has the net effect of an animated GIF:

- You define a series of images that serve as the frames of the animation
- You define how long each of those images should be on the screen
- You define whether the animation should loop back to the beginning after it reaches the end or not

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, rather than encoding all of this in an animated GIF, you instead encode this information in an XML file, stored as a drawable resource.

XML-encoded drawable resources are typically stored in a drawable directory that does *not* contain density information, such as `res/drawable/`. That is because the XML-encoded drawable resources are density-invariant: they behave the same regardless of density. Those, like the `AnimationDrawable`, that refer to other images might well refer to other images that are stored in density-dependent resource directories, but the XML-encoded drawable itself is independent of density.

An `AnimationDrawable` is defined as in XML with a root `<animation-list>` element, containing a series of `<item>` elements for each frame:

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="true">
    <item android:drawable="@drawable/frame1" android:duration="250" />
    <item android:drawable="@drawable/frame2" android:duration="250" />
    <item android:drawable="@drawable/frame3" android:duration="250" />
    <item android:drawable="@drawable/frame4" android:duration="250" />
</animation-list>
```

The root `<animation-list>` element can have an `android:oneshot` attribute, indicating whether the animation should repeat after displaying the last frame (`false`) or stop (`true`).

The `<item>` elements have `android:drawable` attributes pointing to the individual images for the individual frames. Usually these frames are PNG or JPEG files, but you refer to them as drawable resources, using `@drawable` syntax, so Android can find the right image based upon the density (or other characteristics) of the current device. The `<item>` elements also need an `android:duration` attribute, specifying the time in milliseconds that this frame should be on the screen. While the above example has all durations the same, that is not required.

For example, the Android OS uses `AnimationDrawable` resources in a few places. One is for the download icon used in a `Notification` for use with `DownloadManager` and similar situations. That drawable resource – `stat_sys_download.xml` — looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
/* //device/apps/common/res/drawable/status_icon_background.xml
**
** Copyright 2008, The Android Open Source Project
**
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
** Licensed under the Apache License, Version 2.0 (the "License");
** you may not use this file except in compliance with the License.
** You may obtain a copy of the License at
**
**     http://www.apache.org/licenses/LICENSE-2.0
**
** Unless required by applicable law or agreed to in writing, software
** distributed under the License is distributed on an "AS IS" BASIS,
** WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
** See the License for the specific language governing permissions and
** limitations under the License.
*/
-->
<animation-list
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:oneshot="false">
    <item android:drawable="@drawable/stat_sys_download_anim0"
android:duration="200" />
    <item android:drawable="@drawable/stat_sys_download_anim1"
android:duration="200" />
    <item android:drawable="@drawable/stat_sys_download_anim2"
android:duration="200" />
    <item android:drawable="@drawable/stat_sys_download_anim3"
android:duration="200" />
    <item android:drawable="@drawable/stat_sys_download_anim4"
android:duration="200" />
    <item android:drawable="@drawable/stat_sys_download_anim5"
android:duration="200" />
</animation-list>
```

Here, we have a repeating animation (`android:oneshot="false"`), consisting of six frames, each on the screen for 200 milliseconds.

By specifying an `AnimationDrawable` in your `Notification` for its icon, you too can have this sort of animated effect. Of course, the animation is "fire and forget": other than by removing or replacing the `Notification`, you cannot affect the animation in any other way.

Animated GIF Conversion

It may be that you have an animated GIF that you would like to use as the basis for your `AnimationDrawable`. If you have passing familiarity with Ruby, the author of this book has published a Ruby script, named `gif2animdraw`, that automates the conversion.

To use `gif2animdraw`, in addition to the script itself and a Ruby interpreter, you will need the `RMagick`, `slop`, and `builder` gems. Note that `RMagick`, in turn, will require

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

ImageMagick libraries and therefore is a bit more complicated to install than is your ordinary gem.

On Linux environments, you can also `chmod` the script to run it directly; otherwise, you would run it via the `ruby` command.

The script takes four command-line switches:

- `-i` should point to the GIF file to be converted
- `-o` should point to the root output directory, which typically would be a project's `res/` directory
- `-d` should have, as a value, one of the Android density bucket names (e.g., `hdpi`); this will be used as the density for the frames of the GIF
- Optionally, include `--oneshot` to indicate that this should be a one-shot animation, not a repeating one

The results will be:

- A `drawable/` directory underneath your supplied root, containing a file with the same name as the GIF file, but with a `.xml` extension, representing the `AnimationDrawable` itself
- A `drawable-XXXX/` directory, where `XXXX` is your stated density, containing each frame of the animated GIF, as a PNG file, with a sequentially numbered filename based on the GIF's filename

StateListDrawable

Another XML-defined drawable resource, the `StateListDrawable`, is key if you want to have different images when widgets are in different states.

As outlined in the introduction to this chapter, what makes a `Button` visually be a `Button` is its background. To handle different looks for the `Button` background for different states (normal, pressed, disabled, etc.), the standard `Button` background is a `StateListDrawable`, one that looks something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->

<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_window_focused="false" android:state_enabled="true"
        android:drawable="@drawable/btn_default_normal" />
    <item android:state_window_focused="false" android:state_enabled="false"
        android:drawable="@drawable/btn_default_normal_disable" />
    <item android:state_pressed="true"
        android:drawable="@drawable/btn_default_pressed" />
    <item android:state_focused="true" android:state_enabled="true"
        android:drawable="@drawable/btn_default_selected" />
    <item android:state_enabled="true"
        android:drawable="@drawable/btn_default_normal" />
    <item android:state_focused="true"
        android:drawable="@drawable/btn_default_normal_disable_focused" />
    <item
         android:drawable="@drawable/btn_default_normal_disable" />
</selector>
```

The XML has a `<selector>` root element, indicating this is a `StateListDrawable`. The `<item>` elements inside the root describe what Drawable resource should be used if the `StateListDrawable` is being used in some state. For example, if the "window" (think activity or dialog) does not have the focus (`android:state_window_focused="false"`) and the `Button` is enabled (`android:state_enabled="true"`), then we use the `@drawable/btn_default_normal` `Drawable` resource. That resource, as it turns out, is a nine-patch PNG file, described later in this chapter.

Android applies each rule in turn, top-down, to find the `Drawable` to use for a given state of the `StateListDrawable`. The last rule has no `android:state_*` attributes, meaning it is the overall default image to use if none of the other rules match.

So, if you want to change the background of a `Button`, you need to:

- Copy the above resource, found in your Android SDK as `res/drawable/btn_default.xml` inside any of the `platforms/` directories, into your project
- Copy each of the `Button` state nine-patch images into your project
- Modify whichever of those nine-patch images you want, to affect the visual change you seek

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- If need be, tweak the states and images defined in the `StateListDrawable` XML you copied
- Reference the local `StateListDrawable` as the background for your `Button`

The backgrounds of most widgets that have backgrounds by default will use a `StateListDrawable`. Searching a platform version's `res/drawable/` directory for XML files containing `<selector>` elements comes up with a rather long list.

ColorStateList

A `ColorStateList` is analogous to a `StateListDrawable`, in that it defines states and identifies what should be used for a given state. Whereas `StateListDrawable` ties states to drawables, `ColorStateList` ties states to colors. This allows you to, say, change the color of some text based upon whether that text is drawn in a widget that is being pressed, or has the focus, or is disabled. If you tailor the background of a text-based widget using a `StateListDrawable`, you may well wind up tailoring the foreground text using a `ColorStateList`.

While this chapter mentions `ColorStateList`, technically a `ColorStateList` is not a `Drawable`. You do not use it in methods that take drawables or in widget XML attributes that take drawables. Rather, there are other methods and other attributes that take a `ColorStateList`, such as `android:textColor`.

Similarly, while you can define a `ColorStateList` in XML, you do not do so in a `res/drawable/` resource directory, but rather a `res/color/` resource directory. Beyond that, though, a `ColorStateList` XML resource looks a lot like a `StateListDrawable` XML resource, such as this definition of `@android:color/primary_text_dark` from Android 4.4:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_enabled="false" android:color="@android:color/
bright_foreground_dark_disabled"/>
    <item android:state_window_focused="false" android:color="@android:color/
bright_foreground_dark"/>
    <item android:state_pressed="true" android:color="@android:color/
bright_foreground_dark_inverse"/>
    <item android:state_selected="true" android:color="@android:color/
bright_foreground_dark_inverse"/>
    <item android:state_activated="true" android:color="@android:color/
bright_foreground_dark_inverse"/>
    <item android:color="@android:color/bright_foreground_dark"/> <!-- not
selected -->
</selector>
```

Based upon the state, the `ColorStateList` pulls in a separate resource to define the actual color. Those colors, in turn, are defined via `<color>` elements in `res/values/colors.xml` as color resources:

```
    <color name="background_dark">#ff000000</color>
    <color name="background_light">#ffffffff</color>
    <color name="bright_foreground_dark">@android:color/background_light</color>
    <color name="bright_foreground_light">@android:color/background_dark</color>
    <color name="bright_foreground_dark_disabled">#80ffffff</color>
    <color name="bright_foreground_light_disabled">#80000000</color>
    <color name="bright_foreground_dark_inverse">@android:color/
bright_foreground_light</color>
    <color name="bright_foreground_light_inverse">@android:color/
bright_foreground_dark</color>
```

LayerDrawable

A `LayerDrawable` basically stacks a bunch of other drawables on top of each other. Later drawables are drawn on top of earlier drawables, much as later children of a `RelativeLayout` are drawn on top of earlier children.

Typically, you will create a `LayerDrawable` via a `<layer-list>` XML drawable resource.

For example, a `ToggleButton` widget has a `LayerDrawable` as its background:

```
?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
-->

<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+android:id/background"
android:drawable="@android:drawable/btn_default_small" />
    <item android:id="@+android:id/toggle" android:drawable="@android:drawable/
btn_toggle" />
</layer-list>
```

This `LayerDrawable` draws two images on top of each other. One is a standard small button background (`@android:drawable/btn_default_small`). The other is the actual face of the toggle itself — a `StateListDrawable` that uses different images for checked and unchecked states.

In the `<layer-list>`, you can have several `<item>` elements. Each `<item>` element usually will need an `android:drawable` attribute, pointing to the drawable that should be drawn. Optionally, you can assign ID values to the items via `android:id` attributes, much like you would do for widgets in a layout XML resource. Later on, you can call `findDrawableByLayerId()` on the `LayerDrawable` to retrieve an individual `Drawable` representing the layer, given its `android:id` value.

There are also `android:left`, `android:right`, `android:top`, and `android:bottom` attributes, which you can use to provide dimension values to offset an image within the layered set. For example, you could use `android:left` to inset one of the layers by a certain number of pixels (or `dp` or whatever).

By default, the layers in the `LayerDrawable` are scaled to fit the size of whatever `View` is holding them (e.g., the size of the `ToggleButton` using the `LayerDrawable` as a background). To prevent this, you can skip the `android:drawable` attribute, and instead nest a `<bitmap>` element inside the `<item>`, where you can provide an `android:gravity` attribute to control how the image should be handled relative to its containing `View`. We will get more into nested `<bitmap>` elements later in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

TransitionDrawable

A `TransitionDrawable` is a `LayerDrawable` with one added feature: for a two-layer drawable, it can smoothly transition from showing one layer to another on top.

For example, you may have noticed that when you tap-and-hold on a row in a `ListView` that the selector highlight has an animated effect, slowly shifting colors from the color used for a simple click to one signifying that you have long-clicked the row. Android accomplishes this via a `TransitionDrawable`, set up as a `<transition>` XML drawable resource:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->

<transition xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@android:drawable/list_selector_background_pressed"
/>
    <item android:drawable="@android:drawable/
list_selector_background_longpress"  />
</transition>
```

The `TransitionDrawable` object has a `startTransition()` method that you can use, that will have Android smoothly switch from the first drawable to the second. You specify the duration of the transition as a number of milliseconds passed to `startTransition()`. There are also options to reverse the transition, set up more of a cross-fade effect, and the like.

LevelListDrawable

A `LevelListDrawable` is similar in some respects to a `StateListDrawable`, insofar as one specific item from the "list drawable" will be displayed based upon certain conditions. In the case of `StateListDrawable`, the conditions are based upon the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

state of the widget using the drawable (e.g., checked, pressed, disabled). In the case of `LevelListDrawable`, it is merely an integer level.

For example, the status or system bar of your average Android device has an icon indicating the battery charge level. That is actually implemented as a `LevelListDrawable`, via an XML resource containing a root `<level-list>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
/* //device/apps/common/res/drawable/stat_sys_battery.xml
**
** Copyright 2007, The Android Open Source Project
**
** Licensed under the Apache License, Version 2.0 (the "License");
** you may not use this file except in compliance with the License.
** You may obtain a copy of the License at
**
**     http://www.apache.org/licenses/LICENSE-2.0
**
** Unless required by applicable law or agreed to in writing, software
** distributed under the License is distributed on an "AS IS" BASIS,
** WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
** See the License for the specific language governing permissions and
** limitations under the License.
*/
-->

<level-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:maxLevel="4" android:drawable="@android:drawable/
stat_sys_battery_0" />
    <item android:maxLevel="15" android:drawable="@android:drawable/
stat_sys_battery_15" />
    <item android:maxLevel="35" android:drawable="@android:drawable/
stat_sys_battery_28" />
    <item android:maxLevel="49" android:drawable="@android:drawable/
stat_sys_battery_43" />
    <item android:maxLevel="60" android:drawable="@android:drawable/
stat_sys_battery_57" />
    <item android:maxLevel="75" android:drawable="@android:drawable/
stat_sys_battery_71" />
    <item android:maxLevel="90" android:drawable="@android:drawable/
stat_sys_battery_85" />
    <item android:maxLevel="100" android:drawable="@android:drawable/
stat_sys_battery_100" />
</level-list>
```

This `LevelListDrawable` has eight items, whose `android:drawable` attributes point to specific other drawable resources (in this case, standard PNG files with different implementations for different densities). Each `<item>` has an `android:maxLevel` value. When someone calls `setLevel()` on the `Drawable` or `setImageLevel()` on the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`ImageView`, Android will choose the item with the lowest `maxLevel` that meets or exceeds the requested level, and show that. In the case of the battery icon, when the battery level changes, the status bar picks up that change and calls `setImageLevel()` with the battery charge percentage (expressed as an integer from 0–100) — that, in turn, triggers the right PNG file to be displayed.

Another use of `LevelListDrawable` is with a `RemoteViews`, such as for an app widget. The `setImageLevel()` method is "remotable", despite not being directly part of the `RemoteViews` API. Hence, given that you use a `LevelListDrawable` in your app widget's layout, you should be able to use `setInt()` with a method name of `"setImageLevel"` to have the app widget update to display the proper image.

ScaleDrawable and ClipDrawable

A `ScaleDrawable` does pretty much what its name suggests: it scales another drawable. A `ClipDrawable` does pretty much what *its* name suggests: it clips another drawable.

How they do this, and how you control it, requires a bit more explanation.

Like `LevelListDrawable`, `ScaleDrawable` and `ClipDrawable` leverage the `setLevel()` method on `Drawable` (or the `setImageLevel()` method on `ImageView`). Whereas `LevelListDrawable` uses this to choose an individual image out of a set of possible images, `ScaleDrawable` and `ClipDrawable` use the level to control how much an image should be scaled or clipped. For this, they support a range of levels from 0 to 10000.

Scaling

For a level of 0, `ScaleDrawable` will not draw anything. For a level from 1 to 10000, `ScaleDrawable` will scale an image from a configurable minimum size to the bounds of the `View` to which the drawable is applied.

The amount of scaling is determined by `android:scaleHeight` and `android:scaleWidth` attributes:

```
<?xml version="1.0" encoding="utf-8"?>
<scale xmlns:android="http://schemas.android.com/apk/res/android"
  android:drawable="@android:drawable/btn_default"
  android:scaleGravity="left|top"
  android:scaleHeight="50%"
  android:scaleWidth="50%"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The above ScaleDrawable (denoted by the <scale> root element) says that we should scale both height and width of the underlying drawable to 50% of the available space for the drawable, when the level is at its maximum (10000).

Note that you do not have to scale along both dimensions. If, for example, you kept android:scaleWidth but deleted android:scaleHeight, setImageLevel() would control the scaled width of the underlying image (provided via android:drawable) but not the height.

The android:scaleGravity attribute indicates where the scaled image should reside within the available space (the 10000 level, determined by the bounds of the View to which the drawable is applied). The value shown above, center, keeps the image centered within the available space, and shrinks or expands it around the center. A value of left|top would keep the image in the upper-left corner of the space, having the visual effect of moving the lower-right corner based upon the supplied level.

Clipping

Scaling proportionally reduces the height and/or width of an image. Clipping, on the other hand, chops off part of the height or width of the image.

```
<clip xmlns:android="http://schemas.android.com/apk/res/android"
  android:clipOrientation="horizontal"
  android:drawable="@drawable/btn_default_normal"
  android:gravity="left"/>
```

In this sample ClipDrawable (indicated by the <clip> root element), we are going to allow the level to chop off part of the image indicated by the android:drawable attribute. Our android:clipOrientation, set to horizontal, means we are going to chop off part of the width (vertical would have us chop off part of the height). The amount that is going to be chopped off is the level you supply (e.g., setImageLevel()) divided by 10000. Hence, a level of 5000 will chop off 0.5 (a.k.a., 50%) of the image.

Where in the image the clipping occurs is determined by the android:gravity attribute. An android:clipOrientation of horizontal and an android:gravity of left, as in the sample drawable above, means that the left side of the image is retained, and the image will be clipped on the right. Specifying right instead of left would reverse that, clipping the image from the right, while center would clip equally from both sides. There are other gravity values as well, such as top and bottom values to be used with a vertical orientation.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Seeing It In Action

To see these effects, take a look at the Drawable/ScaleClip sample project. This is derived from an earlier example showing how to use ViewPager with PagerTabStrip. In that example, we had 10 tabs, each being a large EditText widget. In this example, we have 2 tabs, "Scale" and "Clip", both using the same layout:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <ImageView
    android:id="@+id/image"
    android:layout_width="150dp"
    android:layout_height="150dp"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="20dp"
    android:scaleType="fitXY"/>

  <SeekBar
    android:id="@+id/level"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_marginBottom="20dp"
    android:layout_marginLeft="20dp"
    android:layout_marginRight="20dp"
    android:max="10000"
    android:progress="10000"/>

</RelativeLayout>
```

This is simply a 150dp square ImageView towards the top of the screen and a SeekBar towards the bottom of the screen. The SeekBar will be used to control the level applied to a ScaleDrawable and ClipDrawable, which is why we have android:max set to 10000. We also have our "progress" (original SeekBar value) set to 10000, so the bar's thumb will be fully slid over to the right at the outset.

The fragments that we will use for the tabs both inherit from a common abstract FragmentBase class:

```
package com.commonsware.android.scaleclip;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.widget.SeekBar;

abstract public class FragmentBase extends Fragment implements
    SeekBar.OnSeekBarChangeListener {
  abstract void setImageBackground(ImageView image);

  private ImageView image=null;

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    setRetainInstance(true);

    View result=inflater.inflate(R.layout.scaleclip, container, false);
    SeekBar bar=((SeekBar)result.findViewById(R.id.level));

    bar.setOnSeekBarChangeListener(this);
    image=(ImageView)result.findViewById(R.id.image);
    setImageBackground(image);
    image.setImageLevel(bar.getProgress());

    return(result);
  }

  @Override
  public void onProgressChanged(SeekBar seekBar, int progress,
                                boolean fromUser) {
    image.setImageLevel(progress);
  }

  @Override
  public void onStartTrackingTouch(SeekBar seekBar) {
    // no-op
  }

  @Override
  public void onStopTrackingTouch(SeekBar seekBar) {
    // no-op
  }
}
```

In `onCreateView()`, we inflate the above layout file, hook up the fragment itself to be the listener for `SeekBar` change events, call the subclass' `setImageBackground()` method to populate the `ImageView` with an image, and set the `ImageView`'s level to be the initial value of the `SeekBar`. When the `SeekBar` value changes, our `onProgressChanged()` method will adjust the level.

The concrete subclasses — `ScaleFragment` and `ClipFragment` — simply populate the `ImageView` with the `ScaleDrawable` and `ClipDrawable` resources shown earlier in this section:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.scaleclip;

import android.widget.ImageView;

public class ScaleFragment extends FragmentBase {
  @Override
  void setImageBackground(ImageView image) {
    image.setImageResource(R.drawable.scale);
  }
}
```

```
package com.commonsware.android.scaleclip;

import android.widget.ImageView;

public class ClipFragment extends FragmentBase {
  @Override
  void setImageBackground(ImageView image) {
    image.setImageResource(R.drawable.clip);
  }
}
```

Those two drawables based their scaling and clipping on `res/drawable-xdpi/btn_default_normal.9.png`. This is a slightly-modified copy of the default button background, and is a nine-patch PNG file. We will discuss nine-patch PNG files later in this chapter — suffice it to say for now that it is a PNG file with rules about how it should be stretched.

Our scale tab starts off showing the full image:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

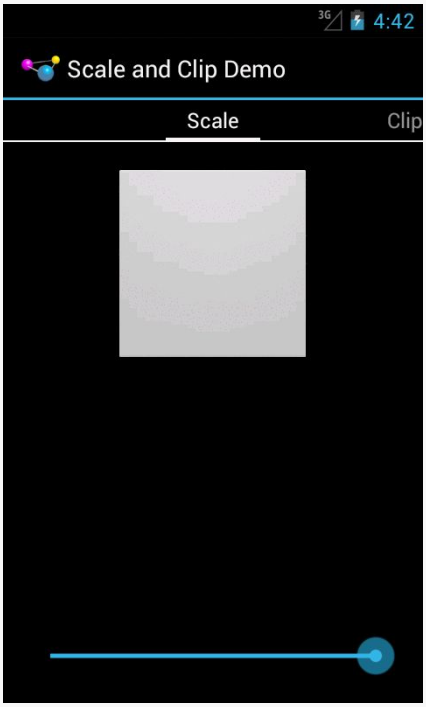


Figure 430: ScaleDrawable, Level of 10000

As we start sliding the `SeekBar` thumb to the left, the image shrinks progressively:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

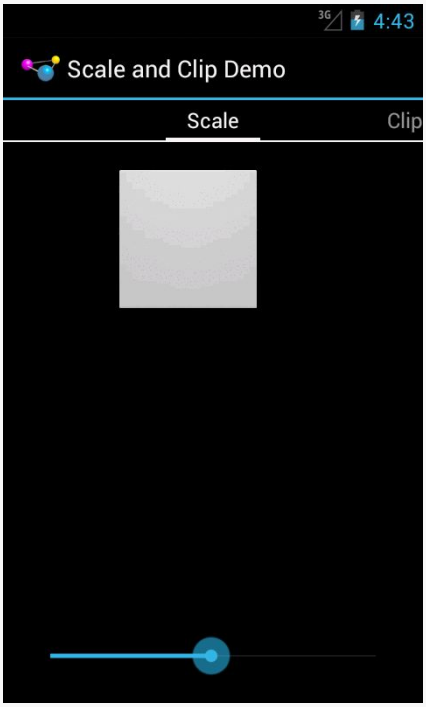


Figure 431: ScaleDrawable, Level of Approximately 5000

It eventually tends towards the 50% level specified in our `android:scaleHeight` and `android:scaleWidth` values:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

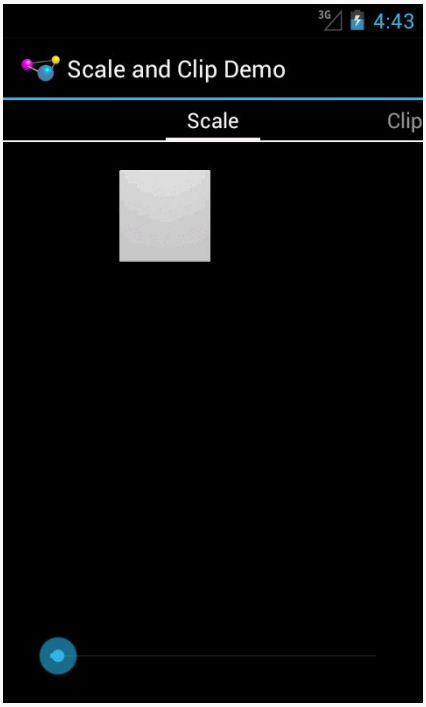


Figure 432: ScaleDrawable, Level of Approximately 100

Sliding it all the way to the left, though, causes the image to vanish.

The `ClipDrawable` starts off looking much like the `ScaleDrawable`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

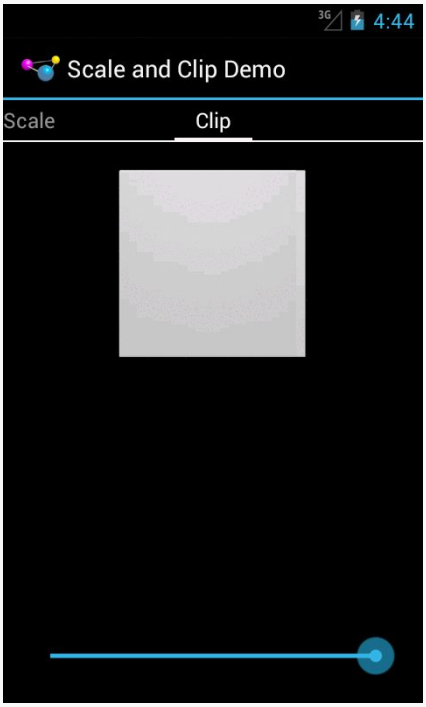


Figure 433: ClipDrawable, Level of 10000

As we slide the `SeekBar` to the left, the right side of the image gets clipped:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

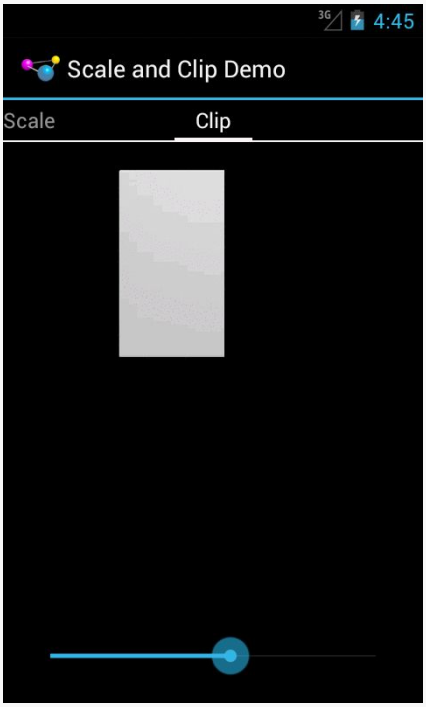


Figure 434: ClipDrawable, Level of Approximately 5000

InsetDrawable

An `InsetDrawable` allows you to apply insets on any side (or all sides) of some other drawable resource. The use case cited in the documentation is "This is used when a View needs a background that is smaller than the View's actual bounds". However, at the present time, nothing in the Android open source code uses this particular type of resource, or even the Java class.

In principle, though, you could have an XML drawable resource that looked like this:

```
<?xml version="1.0" encoding="utf-8"?>
<inset xmlns:android="http://schemas.android.com/apk/res/android"
    android:drawable="@drawable/something_or_another"
    android:insetLeft="20dp"
    android:insetTop="10dp" />
```

When used as the background for some `View`, for example, Android would pull in the `something_or_another` resource and effectively add `20dp` of left margin and `10dp` of top margin on the background when calculating its size and drawing it on the screen.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

ShapeDrawable

Far and away the most complex of the XML drawable formats is the `ShapeDrawable`. It gives you what amounts to a very tiny subset of SVG, for creating simple vector art shapes.

The root element of a `ShapeDrawable` resource is `<shape>`, which may have child elements, along with attributes, to configure what gets rendered on the screen when the drawable is applied.

This section will review the elements and attributes available to you, with sample drawables (and screenshots) culled from the Drawable/Shape sample project.

This is a "sampler" project, designed to depict a number of `ShapeDrawables`. To accomplish this, we will use action bar tabs. Our activity (`MainActivity`) has a pair of static `int` arrays, one pointing at string resources to use for tab captions, the other pointing at corresponding drawable resources:

```
package com.commonsware.android.shape;

import android.app.ActionBar;
import android.app.ActionBar.Tab;
import android.app.ActionBar.TabListener;
import android.app.Activity;
import android.app.FragmentTransaction;
import android.os.Bundle;
import android.widget.ImageView;

public class MainActivity extends Activity implements TabListener {
  private static final int TABS[]= { R.string.solid, R.string.gradient,
      R.string.border, R.string.rounded, R.string.ring,
      R.string.layered };
  private static final int DRAWABLES[]= { R.drawable.rectangle,
      R.drawable.gradient, R.drawable.border, R.drawable.rounded,
      R.drawable.ring, R.drawable.layered };
  private ImageView image=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    image=(ImageView)findViewById(R.id.image);

    ActionBar bar=getActionBar();
    bar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

    for (int i=0; i < TABS.length; i++) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      bar.addTab(bar.newTab().setText(getString(TABS[i]))
                 .setTabListener(this));
    }
  }

  @Override
  public void onTabSelected(Tab tab, FragmentTransaction ft) {
    image.setImageResource(DRAWABLES[tab.getPosition()]);
  }

  @Override
  public void onTabUnselected(Tab tab, FragmentTransaction ft) {
    // no-op
  }

  @Override
  public void onTabReselected(Tab tab, FragmentTransaction ft) {
    // no-op
  }
}
```

In `onCreate()`, we toggle the `ActionBar` into tab-navigation mode, then iterate over the arrays and add one tab per element.

Our layout is an `ImageView`, named `image`, centered on the screen, taking up 80% of the horizontal space, plus has `20dp` of top and bottom margin:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/LinearLayout1"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal"
  android:gravity="center"
  android:weightSum="10">

    <ImageView
      android:id="@+id/image"
      android:src="@drawable/rectangle"
      android:layout_width="0dp"
      android:layout_height="match_parent"
      android:layout_marginTop="20dp"
      android:layout_marginBottom="20dp"
      android:layout_gravity="center"
      android:layout_weight="8"/>

</LinearLayout>
```

In our activity's `onTabSelected()` — implemented because the activity is the `TabListener` for our tabs — we get the position of our tab and fill in the appropriate drawable into the `ImageView`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Given that, let's take a look at how to construct a `ShapeDrawable`, along with some sample drawables.

<shape>

Your root element, not surprisingly, is `<shape>`.

The primary thing that you will define on the `<shape>` element is the redundantly-named `android:shape` attribute, to define what sort of shape you want:

- `line` (a shape with no interior)
- `oval` (also for ellipses)
- `rectangle` (including rounded rectangles)
- `ring` (for partially-filled circles)

There are some other attributes available on `<shape>` for a `ring`, which we will examine later in this chapter.

<solid>

Your shape will usually require some sort of fill, to say what color goes in the shape. There are two types of fills: solid and gradient.

For a solid fill, add a `<solid>` child element to the `<shape>`, with an `android:color` attribute indicating what color to use. As with most places in Android, this can either be a literal color or a reference to a color resource.

So, for example, we can specify a solid red rectangle as:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">
  <solid android:color="#FFAA0000"/>
</shape>
```

This gives us the following visual result:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

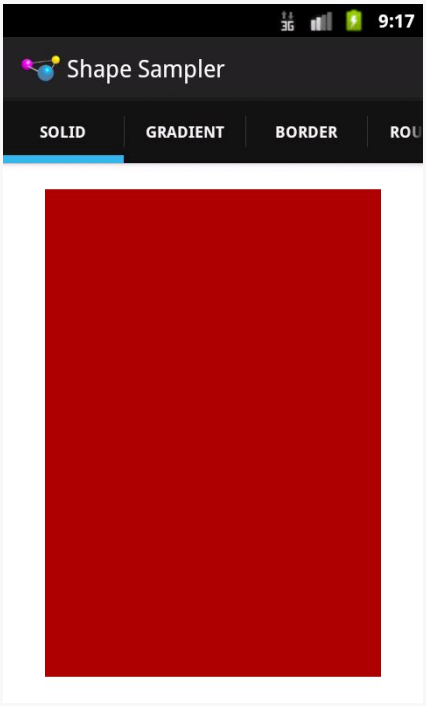


Figure 435: ShapeDrawable, Solid Red Rectangle

<gradient>

Your alternative fill is a gradient. The nice thing about gradients with `ShapeDrawable` is that they are generated at runtime from the specifications in the `ShapeDrawable`, and therefore will be smooth. Gradients that appear in PNG files and the like, if stretched, will tend to have a banding effect.

Gradient fills are defined via a `<gradient>` child element of the `<shape>` element.

The simplest way to set up a gradient is to use three attributes:

- `android:startColor` and `android:endColor`, to specify the starting and ending colors of the gradient, respectively, and
- `android:angle`, to specify what direction the gradient "flows" in

The angle must be a multiple of 45 degrees. 0 degrees is left-to-right, 90 degrees is bottom-to-top, 180 degrees is right-to-left, and 270 degrees is top-to-bottom.

So, for example, we could change our rectangle to have a gradient fill, from red to blue, with red at the top, via:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">

  <gradient
    android:angle="270"
    android:endColor="#FF0000FF"
    android:startColor="#FFFF0000"/>

</shape>
```

That gives us:

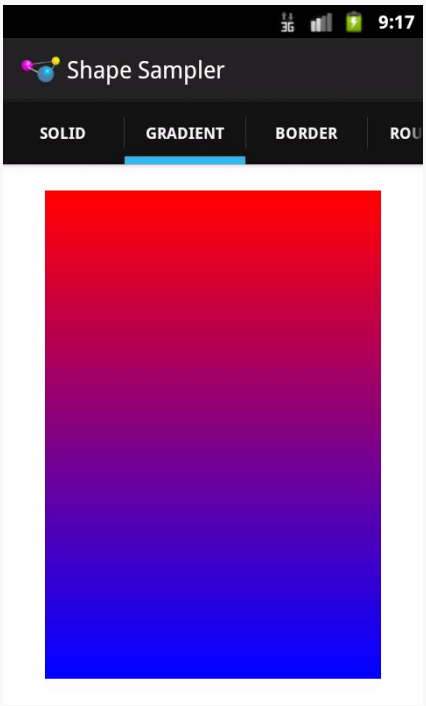


Figure 436: ShapeDrawable, Gradient Fill Rectangle

We will examine some other gradient options in the section on rings, later in this chapter.

<stroke>

If you want a separate color for a border around your shape, you can use the `<stroke>` element, as a child of the `<shape>` element, to configure one.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

There are four attributes that you can declare. The two that you will probably always use are `android:color` (to indicate the color of the border) and `android:width` (to indicate the thickness of the border). By default, using just those two will give you a solid line around the edge of your shape.

If you would prefer a dashed border, you can add in `android:dashWidth` (to indicate how long each dash segment should be) and `android:dashGap` (to indicate how long the gaps between dash segments should be).

So, for example, we can add a dashed border to our gradient rectangle via a suitable `<stroke>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">

  <gradient
    android:angle="270"
    android:endColor="#FF0000FF"
    android:startColor="#FFFF0000"/>

  <stroke
    android:width="2dp"
    android:dashGap="4dp"
    android:dashWidth="20dp"
    android:color="#FF000000"/>

</shape>
```

This gives us:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

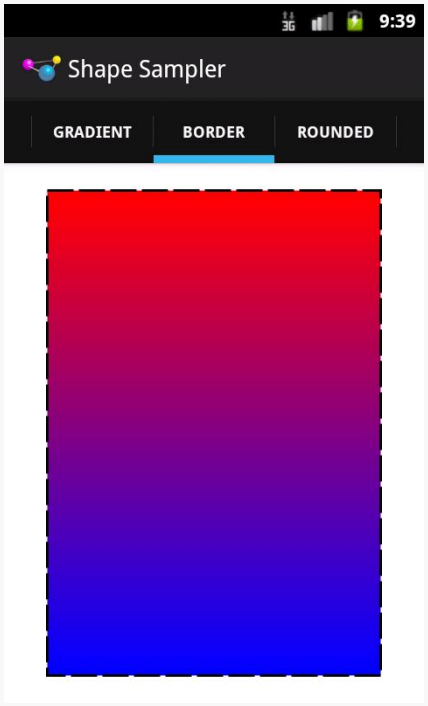


Figure 437: ShapeDrawable, Gradient Fill Rectangle with Dashed Border

<corners>

If we are implementing a `rectangle` shape, but we really want it to be a rounded rectangle, we can add a `<corners>` element as a child of the `<shape>` element. You can specify the radius to apply to the corners, either for all corners (e.g., `android:radius`), or for individual corners (e.g., `android:topLeftRadius`). Here, "radius" basically means the size of the circle that should implement the corner, where a radius of `0dp` would indicate the default square corner.

So, if we wanted to add rounded corners to our gradient-filled, dash-outlined rectangle, we could use this:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">

  <gradient
    android:angle="270"
    android:endColor="#FF0000FF"
    android:startColor="#FFFF0000"/>

  <stroke
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:dashGap="4dp"
    android:dashWidth="20dp"
    android:width="2dp"
    android:color="#FF000000"/>

  <corners android:radius="8dp"/>

</shape>
```

This gives us the following:

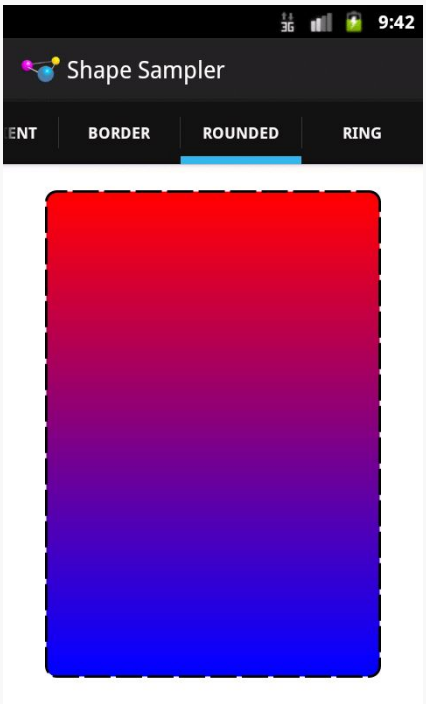


Figure 438: ShapeDrawable, Gradient Fill Rounded Rectangle with Dashed Border

<padding> and <size>

There are also `<padding>` and `<size>` elements that you can add, that specify padding to put on the various sizes and the overall size of the drawable. More often than not, you would actually handle this on the `ImageView` or other widget that is using your drawable, but if you would prefer to define those things in the drawable itself, you are welcome to do so.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Put a Ring On It

Rings are a bit more complicated, in large part because they are not completely filled. With a ring, the "fill" is filling what goes in the ring itself, not the "hole" in the center of the ring. This means that we need to teach Android more about how that "hole" is supposed to be set up.

To do that, we need to provide two pieces of information:

1. How big the inner radius should be, where by "inner radius" Android means "the radius of the hole"
2. How thick the ring should be

The ring will then be drawn based upon that inner radius and thickness.

You might wonder, "well, where does the size of the actual drawable come into play?" After all, if we specify an inner radius of `20dp` and a thickness of `10dp`, that would give us an outer radius of `30dp`, for a total width of `60dp`... regardless of how big the actual drawable is.

And that is completely correct.

However, for both the inner radius and the thickness, you have two choices of how to specify their values:

1. As actual sizes (dimensions or references to dimension resources)
2. As ratios to the overall drawable width (defined by `<size>` or the widget that is using the drawable)

This gives us four total attributes to choose from, to be placed on the `<shape>` element for `ring` drawables:

1. `android:innerRadius`
2. `android:innerRadiusRatio`
3. `android:thickness`
4. `android:thicknessRatio`

Therefore, if you want the ring's size to be based on the size of the drawable, you would use `innerRadiusRatio`, `thicknessRatio`, or both.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The other thing about rings is that they are round. Hence, a default linear gradient fill — going from one side of the drawable to another – may not be what you really want. You can control the type of gradient fill to use via the `android:type` attribute on the `<gradient>` element. There are three possible values:

1. `linear` (the default behavior)
2. `radial`, where the gradient starts from the center (or another point that you define) and changes color from that center to the edges
3. `sweep`, where the gradient revolves clockwise in a circle, starting from whatever `android:angle` you specify (or `0`, meaning "east", as the default)

So, for example, take a look at the following `ShapeDrawable`:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:innerRadiusRatio="3"
  android:shape="ring"
  android:thickness="15dp"
  android:useLevel="false">

  <gradient
    android:centerColor="#4c737373"
    android:endColor="#ff9933CC"
    android:startColor="#4c737373"
    android:type="sweep"/>

</shape>
```

Here, we:

- Declare that our shape is a `ring`
- Indicate that the distance between the inner radius and the outer radius of the ring should be `15dp`
- Indicate that there is a 3:1 ratio between the width of the image and the radius of the "hole" in the ring
- Indicate that the fill should be a gradient that sweeps clockwise from the default angle of 0
- Indicate that the first half of the gradient (start to center) should remain a constant color
- Indicate that the second half of the gradient (center to end) should change color from gray to purple

We also have `android:useLevel="false"` in the `<shape>` element. For unknown reasons, this is required for rings but not for other types of shapes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This gives us:

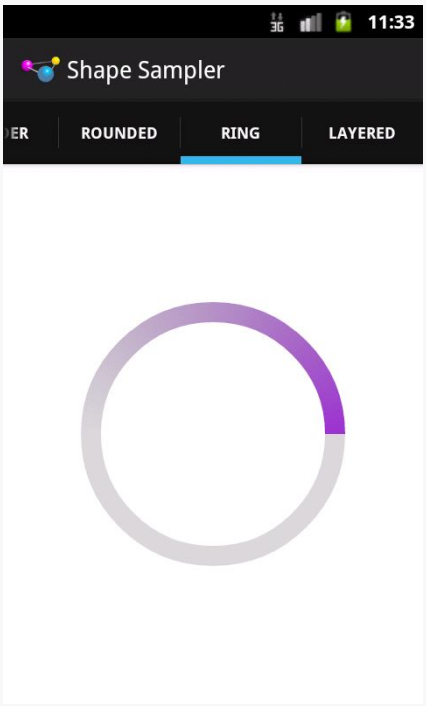


Figure 439: ShapeDrawable, Ring with Gradient Fill

BitmapDrawable

Having an XML drawable format named `BitmapDrawable` may seem like a contradiction in terms. However, `BitmapDrawable` is not an XML representation of a bitmap, but rather an XML representation of operations to perform on an actual bitmap.

The big thing that `BitmapDrawable` offers is `android:tileMode`, which turns a single bitmap into a repeating bitmap. The bitmap is tiled, horizontally and vertically, using a tiling mode that you specify.

This is demonstrated in the `Drawable/TileMode` sample project.

Our activity's layout is just a `LinearLayout`, set to fill the screen:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/widget"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal">

</LinearLayout>
```

Our activity populates action bar tabs, where it applies a particular background image to the `LinearLayout` (known as `R.id.widget`) based on the selected tab:

```
package com.commonsware.android.tilemode;

import android.app.ActionBar;
import android.app.ActionBar.Tab;
import android.app.ActionBar.TabListener;
import android.app.Activity;
import android.app.FragmentTransaction;
import android.os.Bundle;
import android.view.View;

public class MainActivity extends Activity implements TabListener {
  private static final int TABS[]= { R.string._default, R.string.clamp,
      R.string.repeat, R.string.mirror };
  private static final int DRAWABLES[]= { R.drawable._default,
      R.drawable.clamp, R.drawable.repeat, R.drawable.mirror };
  private View widget=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    widget=findViewById(R.id.widget);

    ActionBar bar=getActionBar();
    bar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

    for (int i=0; i < TABS.length; i++) {
      bar.addTab(bar.newTab().setText(getString(TABS[i]))
                    .setTabListener(this));
    }
  }

  @Override
  public void onTabSelected(Tab tab, FragmentTransaction ft) {
    widget.setBackgroundResource(DRAWABLES[tab.getPosition()]);
  }

  @Override
  public void onTabUnselected(Tab tab, FragmentTransaction ft) {
    // no-op
  }

  @Override
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  public void onTabReselected(Tab tab, FragmentTransaction ft) {
    // no-op
  }
}
```

The `res/drawable/_default.xml` resource, used on the first tab, is an unadorned `BitmapDrawable` resource, where our `<bitmap>` element simply has an `android:src` attribute pointing to a bitmap to be used for this `BitmapDrawable`:

```
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
  android:src="@drawable/hatch"/>
```

Since we have not specified a tile mode, the image is stretched to fill the size of our `LinearLayout` when serving as its background:

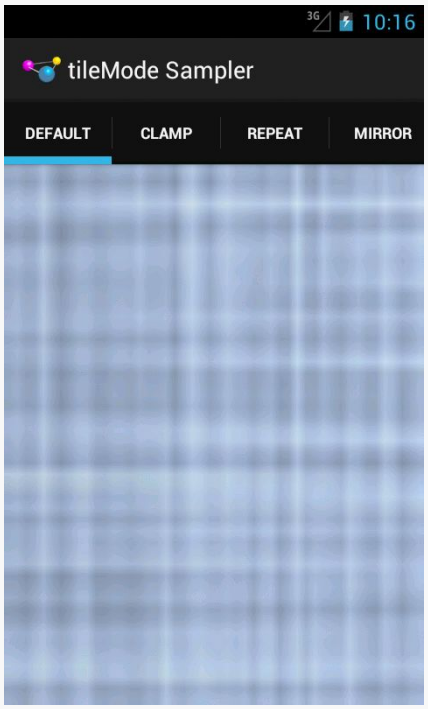


Figure 440: BitmapDrawable, Without android:tileMode

The `res/drawable/clamp.xml` resource, used on the second tab, adds `android:tileMode="clamp"`:

```
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
  android:src="@drawable/hatch"
  android:tileMode="clamp"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This causes the right-most column of pixels and the bottom-most column of pixels to be repeated to fill the available space:

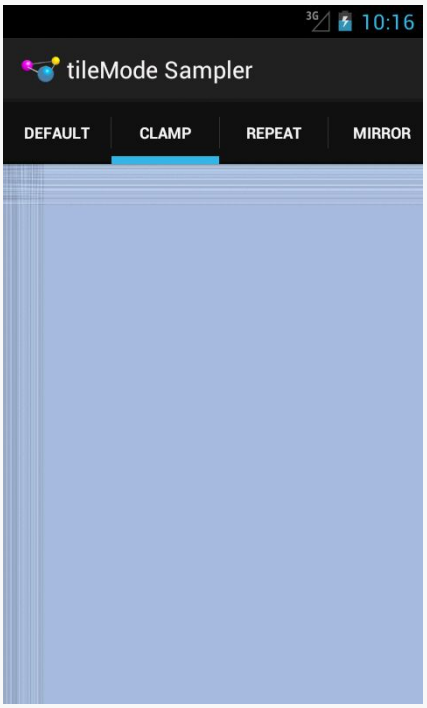


Figure 441: BitmapDrawable, Clamped

Zooming in on the upper-left portion of our `LinearLayout` demonstrates this:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

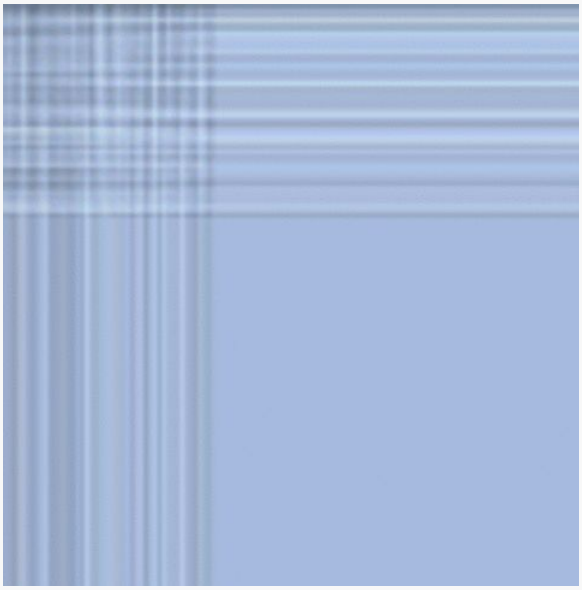


Figure 442: Portion of BitmapDrawable, Clamped

The `res/drawable/repeat.xml` resource, used on the third tab, employs `android:tileMode="repeat"`:

```
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
  android:src="@drawable/hatch"
  android:tileMode="repeat"/>
```

Here, the image is simply repeated *in toto* to fill the available space, rather than only its lower-right edges:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

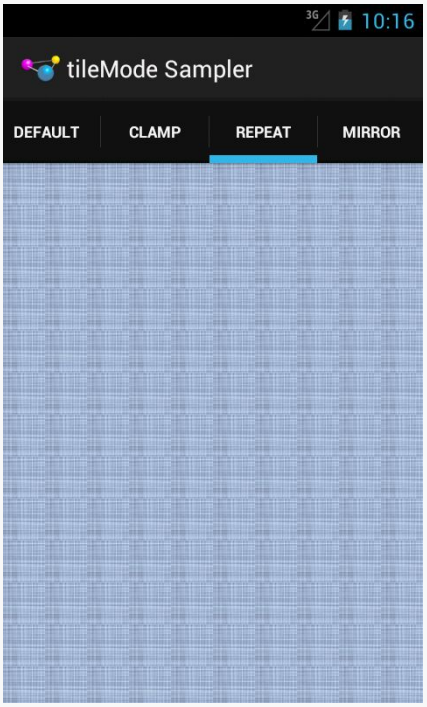


Figure 443: BitmapDrawable, Repeated

Zooming in on an arbitrary chunk of the `LinearLayout` shows this effect:

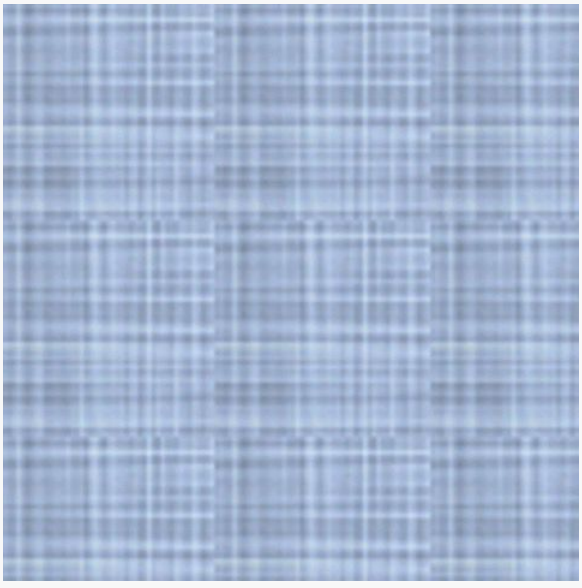


Figure 444: Portion of BitmapDrawable, Repeated

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `res/drawable/mirror.xml` resource, used on the fourth tab, uses `android:tileMode="mirror"`:

```
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
  android:src="@drawable/hatch"
  android:tileMode="mirror"/>
```

Here, the image is repeated, but alternately mirrored along the repeating axis. So, it is flipped horizontally for each repeat along the horizontal axis, and it is flipped vertically for each repeat along the vertical axis:

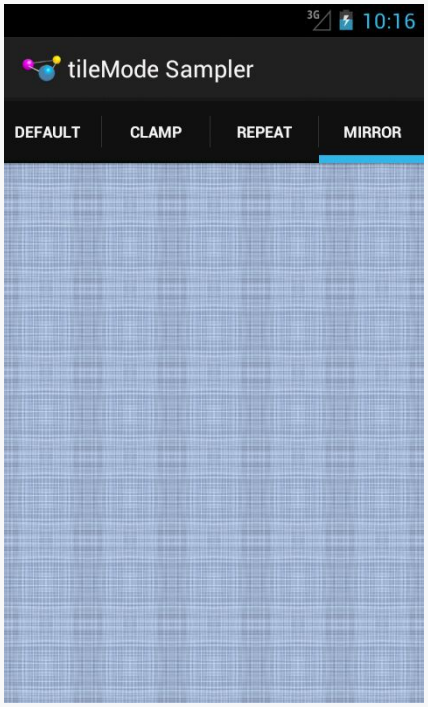


Figure 445: BitmapDrawable, Mirrored

Zooming in on an arbitrary chunk of the `LinearLayout` shows this effect:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

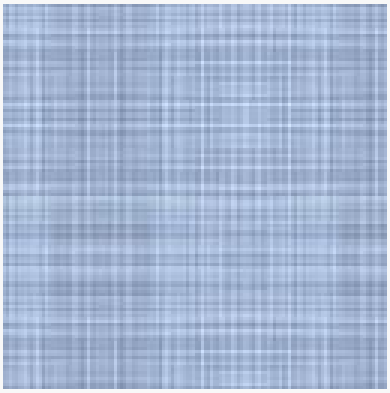


Figure 446: Portion of BitmapDrawable, Mirrored

Composite Drawables

Let's say that we wanted to have a pair of `ShapeDrawable` images, one superimposed on another. Since a single `ShapeDrawable` defines only one shape, we would need something else to assist with stacking the images.

One possibility would be to use a `LayerDrawable`, creating three total resources:

1. The first `ShapeDrawable`, in its own resource file
2. The second `ShapeDrawable`, in its own resource file
3. The `LayerDrawable`, holding references to the two `ShapeDrawable` resources

And this will certainly work. But you have an alternative: put all of it into a single drawable resource.

An `android:drawable` attribute in an `<item>` element can be replaced by child elements representing another drawable structure. Hence, rather than having a `LayerDrawable` with two `<item>` elements pointing to other drawable resources, we could have those same `<item>` elements *contain* the other drawable XML structures, and thereby cut our number of files from 3 to 1.

For example, we could have something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">

  <item>
    <shape android:shape="rectangle">
      <gradient
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        android:angle="270"
        android:endColor="#FF0000FF"
        android:startColor="#FFFF0000"/>

      <stroke
        android:dashGap="4dp"
        android:dashWidth="20dp"
        android:width="2dp"
        android:color="#FF000000"/>

      <corners android:radius="8dp"/>
    </shape>
  </item>
  <item>
    <shape
      android:innerRadiusRatio="3"
      android:shape="ring"
      android:thickness="15dp"
      android:useLevel="false">
      <gradient
        android:endColor="#FFFFFFFF"
        android:startColor="#ff000000"
        android:type="sweep"/>
    </shape>
  </item>

</layer-list>
```

This is a `LayerDrawable`, layering two `ShapeDrawable` structures. The first `ShapeDrawable` is our dash-bordered, gradient-filled, rounded rectangle from before. The second `ShapeDrawable` is a `ring` with a simple gradient sweep fill, from black to white.

This gives us:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

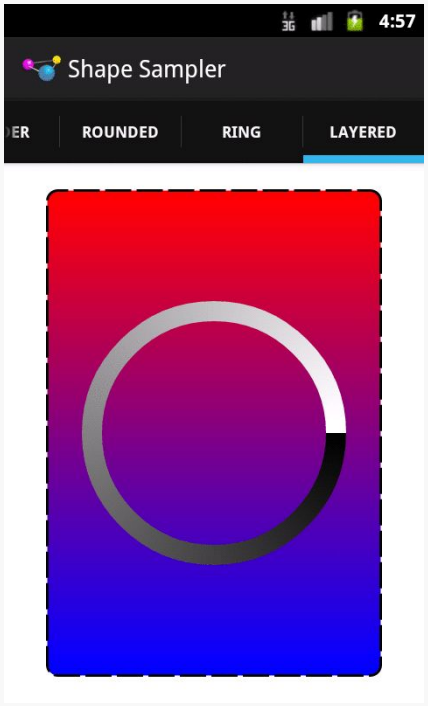


Figure 447: Composite Drawable

Hence, any of the drawable XML structures other than `ShapeDrawable` can, in their `<item>` elements, hold any drawable XML structure, instead of pointing to another separate resource.

Android uses this trick as well. For example, the stock `ProgressBar` image is based off of a `LayerDrawable` wrapped around three `ShapeDrawable` structures:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->

<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    <item android:id="@android:id/background">
        <shape>
            <corners android:radius="5dip" />
            <gradient
                    android:startColor="#ff9d9e9d"
                    android:centerColor="#ff5a5d5a"
                    android:centerY="0.75"
                    android:endColor="#ff747674"
                    android:angle="270"
            />
        </shape>
    </item>

    <item android:id="@android:id/secondaryProgress">
        <clip>
            <shape>
                <corners android:radius="5dip" />
                <gradient
                        android:startColor="#80ffd300"
                        android:centerColor="#80ffb600"
                        android:centerY="0.75"
                        android:endColor="#a0ffcb00"
                        android:angle="270"
                />
            </shape>
        </clip>
    </item>

    <item android:id="@android:id/progress">
        <clip>
            <shape>
                <corners android:radius="5dip" />
                <gradient
                        android:startColor="#ffffd300"
                        android:centerColor="#ffffb600"
                        android:centerY="0.75"
                        android:endColor="#ffffcb00"
                        android:angle="270"
                />
            </shape>
        </clip>
    </item>

</layer-list>
```

We will get into how this works with a `ProgressBar` in a separate chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A Stitch In Time Saves Nine

Most of the types of non-traditional drawable resources you can create in Android are described in XML... but not all.

As you read through the Android documentation, you no doubt ran into references to "nine-patch" or "9-patch" and wondered what Android had to do with quilting. Rest assured, you will not need to take up needlework to be an effective Android developer.

If, however, you are looking to create backgrounds for resizable widgets, like a `Button`, you may wish to work with nine-patch images.

As the Android documentation states, a nine-patch is "a PNG image in which you define stretchable sections that Android will resize to fit the object at display time to accommodate variable sized sections, such as text strings". By using a specially-created PNG file, Android can avoid trying to use vector-based formats (e.g., `ShapeDrawable`) and their associated overhead when trying to create a background at runtime. Yet, at the same time, Android can still resize the background to handle whatever you want to put inside of it, such as the text of a `Button`.

In this section, we will cover some of the basics of nine-patch graphics, including how to customize and apply them to your own Android layouts.

The Name and the Border

Nine-patch graphics are PNG files whose names end in `.9.png`. This means they can be edited using normal graphics tools, but Android knows to apply nine-patch rules to their use.

What makes a nine-patch graphic different than an ordinary PNG is a one-pixel-wide border surrounding the image. When drawn, Android will remove that border, showing only the stretched rendition of what lies inside the border. The border is used as a control channel, providing instructions to Android for how to deal with stretching the image to fit its contents.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Padding and the Box

Along the right and bottom sides, you can draw one-pixel-wide black lines to indicate the "padding box". Android will stretch the image such that the contents of the widget will fit inside that padding box.

For example, suppose we are using a nine-patch as the background of a `Button`. When you set the text to appear in the button (e.g., "Hello, world!"), Android will compute the size of that text, in terms of width and height in pixels. Then, it will stretch the nine-patch image such that the text will reside inside the padding box. What lies outside the padding box forms the border of the button, typically a rounded rectangle of some form.

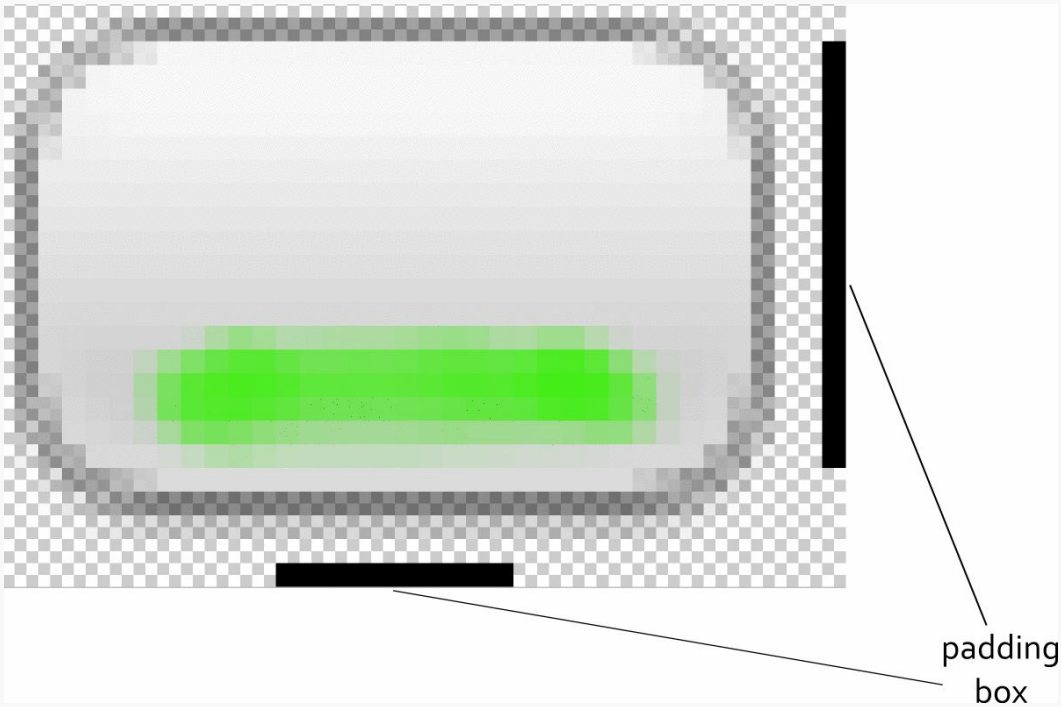


Figure 448: The padding box, as shown by a set of control lines to the right and bottom of the stretchable image

Stretch Zones

To tell Android where on the image to actually do the stretching, draw one-pixel-wide black lines on the top and left sides of the image. Android will scale the graphic only in those areas — areas outside the stretch zones are not stretched.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Perhaps the most common pattern is the center-stretch, where the middle portions of the image on both axes are considered stretchable, but the edges are not:

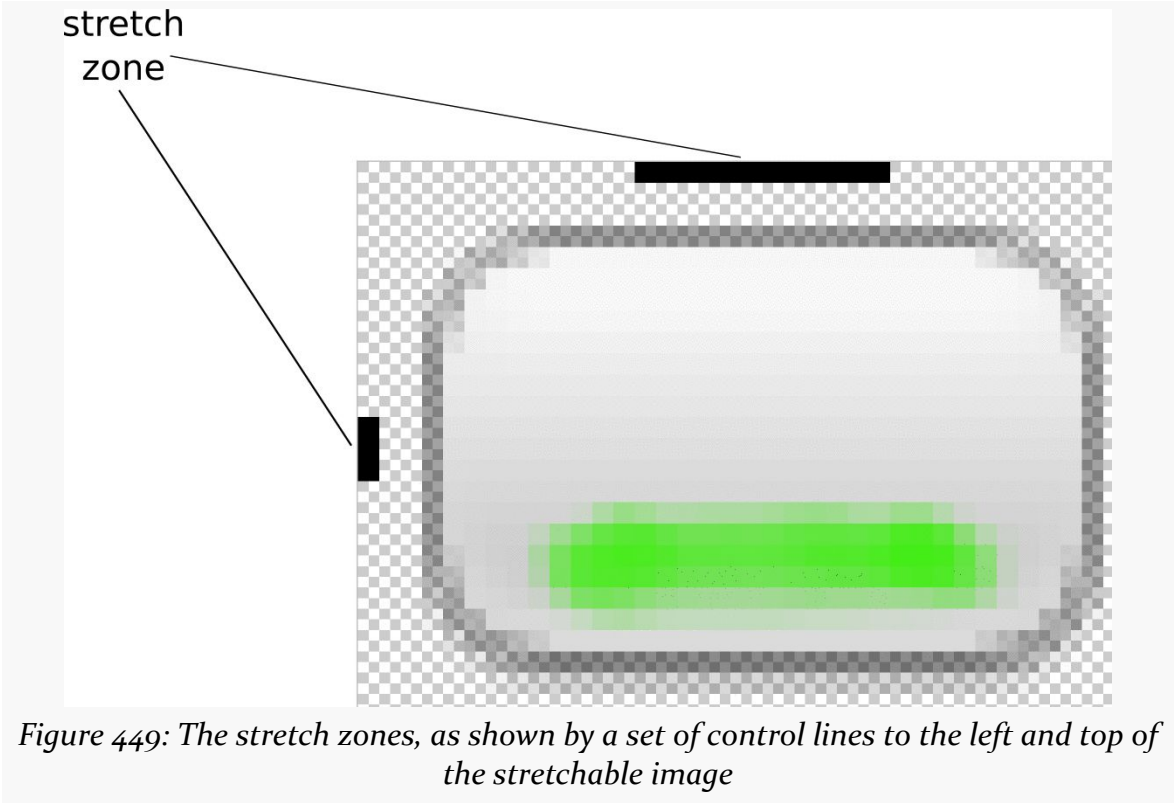


Figure 449: The stretch zones, as shown by a set of control lines to the left and top of the stretchable image

Here, the stretch zones will be stretched just enough for the contents to fit in the padding box. The edges of the graphic are left unstretched.

Some additional rules to bear in mind:

1. If you have multiple discrete stretch zones along an axis (e.g., two zones separated by whitespace), Android will stretch both of them but keep them in their current proportions. So, if the first zone is twice as wide as the second zone in the original graphic, the first zone will be twice as wide as the second zone in the stretched graphic.
2. If you leave out the control lines for the padding box, it is assumed that the padding box and the stretch zones are one and the same.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tooling

To experiment with nine-patch images, you may wish to use the `draw9patch` program, found in the `tools/` directory of your SDK installation:

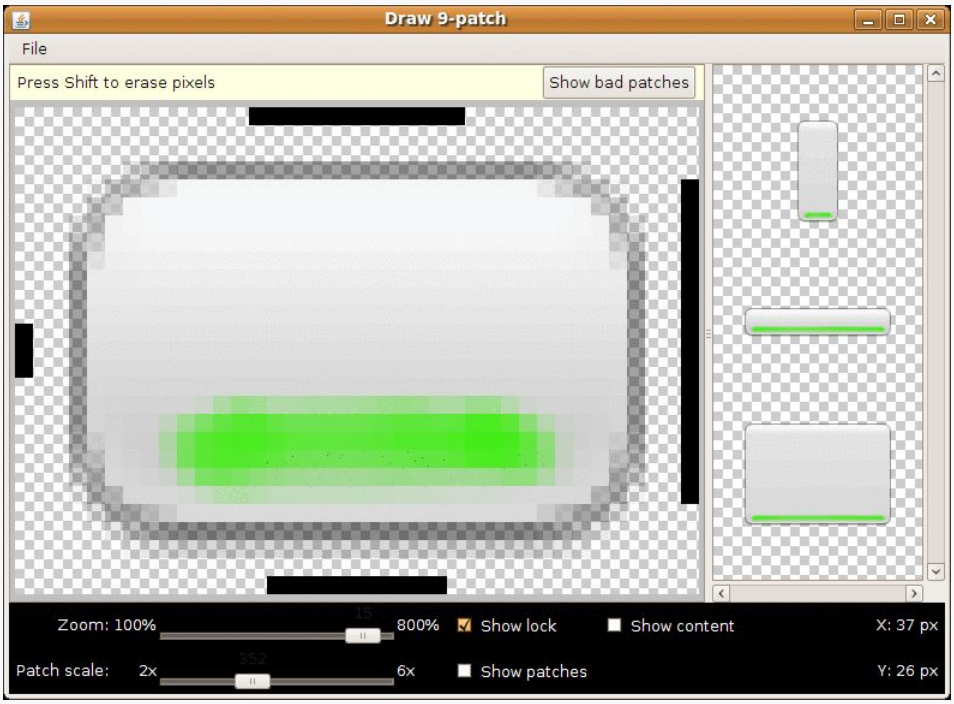


Figure 450: The draw9patch tool

Neither Android Studio nor Eclipse, at the present time, have a built-in version of `draw9patch`, so IDE users will need to run the standalone copy from their SDK installation.

While a regular graphics editor would allow you to draw any color on any pixel, `draw9patch` limits you to drawing or erasing pixels in the control area. If you attempt to draw inside the main image area itself, you will be blocked.

On the right, you will see samples of the image in various stretched sizes, so you can see the impact as you change the stretchable zones and padding box.

While this is convenient for working with the nine-patch nature of the image, you will still need some other graphics editor to create or modify the body of the image itself. For example, the image shown above, from the `Drawable/NinePatch` project, is

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

a modified version of a nine-patch graphic from the SDK's `ApiDemos`, where the GIMP was used to add the neon green stripe across the bottom portion of the image.

Using Nine-Patch Images

Nine-patch images are most commonly used as backgrounds, as illustrated by the following layout from the Drawable/NinePatch sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TableLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1"
  >
    <TableRow
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
    >
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:text="Horizontal:"
      />
      <SeekBar android:id="@+id/horizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
      />
    </TableRow>
    <TableRow
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
    >
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:text="Vertical:"
      />
      <SeekBar android:id="@+id/vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
      />
    </TableRow>
  </TableLayout>
  <LinearLayout
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
    <Button android:id="@+id/resize"
      android:layout_width="96px"
      android:layout_height="96px"
      android:text="Hi!"
      android:textSize="5pt"
      android:background="@drawable/button"
    />
  </LinearLayout>
</LinearLayout>
```

Here, we have two `SeekBar` widgets, labeled for the horizontal and vertical axes, plus a `Button` set up with our nine-patch graphic as its background (`android:background = "@drawable/button"`).

The `NinePatchDemo` activity then uses the two `SeekBar` widgets to let the user control how large the button should be drawn on-screen, starting from an initial size of `64px` square:

```
package com.commonsware.android.ninepatch;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.LinearLayout;
import android.widget.SeekBar;

public class NinePatchDemo extends Activity {
  SeekBar horizontal=null;
  SeekBar vertical=null;
  View thingToResize=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    thingToResize=findViewById(R.id.resize);

    horizontal=(SeekBar)findViewById(R.id.horizontal);
    vertical=(SeekBar)findViewById(R.id.vertical);

    horizontal.setMax(144); // 240 less 96 starting size
    vertical.setMax(144);  // keep it square @ max

    horizontal.setOnSeekBarChangeListener(h);
    vertical.setOnSeekBarChangeListener(v);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }

  SeekBar.OnSeekBarChangeListener h=new SeekBar.OnSeekBarChangeListener() {
    public void onProgressChanged(SeekBar seekBar,
                                  int progress,
                                  boolean fromTouch) {
      ViewGroup.LayoutParams old=thingToResize.getLayoutParams();
      ViewGroup.LayoutParams current=new LinearLayout.LayoutParams(64+progress,
                                                                   old.height);

      thingToResize.setLayoutParams(current);
    }

    public void onStartTrackingTouch(SeekBar seekBar) {
      // unused
    }

    public void onStopTrackingTouch(SeekBar seekBar) {
      // unused
    }
  };

  SeekBar.OnSeekBarChangeListener v=new SeekBar.OnSeekBarChangeListener() {
    public void onProgressChanged(SeekBar seekBar,
                                  int progress,
                                  boolean fromTouch) {
      ViewGroup.LayoutParams old=thingToResize.getLayoutParams();
      ViewGroup.LayoutParams current=new LinearLayout.LayoutParams(old.width,
                                                                   64+progress);

      thingToResize.setLayoutParams(current);
    }

    public void onStartTrackingTouch(SeekBar seekBar) {
      // unused
    }

    public void onStopTrackingTouch(SeekBar seekBar) {
      // unused
    }
  };
}
```

The result is an application that can be used much like the right pane of `draw9patch`, to see how the nine-patch graphic looks on an actual device or emulator in various sizes:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

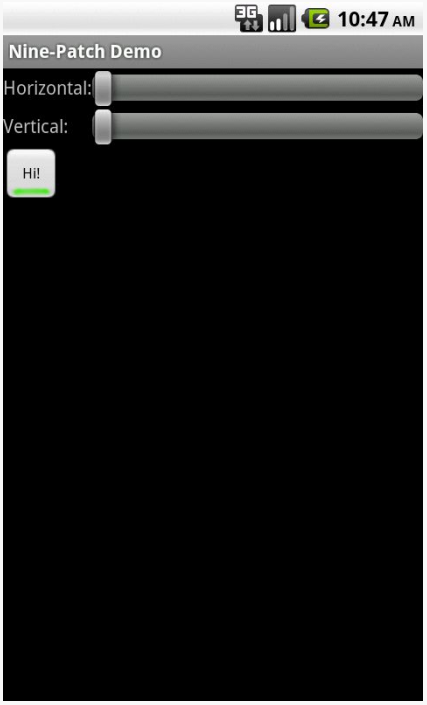


Figure 451: The NinePatch sample project, in its initial state

Figure 452: The NinePatch sample project, after making it bigger horizontally

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

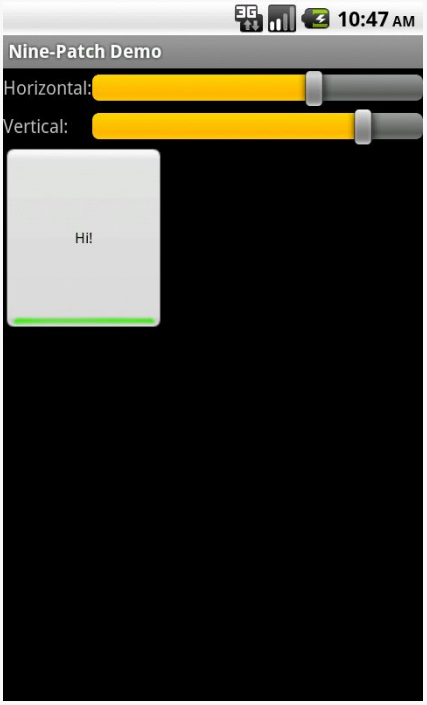


Figure 453: The NinePatch sample application, after making it bigger in both dimensions

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Mapping with Maps V2

One of Google's most popular services — after search, of course – is Google Maps, where you can find everything from the nearest pizza parlor to directions from New York City to San Francisco (only 2,905 miles!) to street views and satellite imagery.

Android has had mapping capability from the beginning, with an API available to us as developers to bake maps into our apps. However, as we will see shortly, that original API was getting a bit stale.

In December 2012, Google released a long-awaited update to the mapping capabilities available to Android app developers. The original mapping solution, now known as the Maps V1, worked but had serious limitations. The new mapping solution, known as Maps V2, offers greater power and greater ease of handling common situations, though it too has its rough edges.

Prerequisites

Understanding this chapter requires that you have read the core chapters, along with the chapter on drawables. Many of the samples also employ list navigation in the action bar — while this is not needed to use Maps V2, knowing that material will help you understand a bit of how the samples work. Also, one of the samples involves location tracking, and another of the samples involves the use of the animator framework.

This chapter also makes the occasional reference back to Maps V1 for comparisons, mostly for the benefit of developers already familiar with Maps V1 and looking to migrate to Maps V2. However, prior experience with Maps V1 is not necessary to understand this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

One section involves the use of Picasso, covered in the chapter on Internet access.

A Brief History of Mapping on Android

Back in the dawn of Android, we were given the Maps SDK add-on. This would allow us to load a firmware-hosted mapping library into our applications, then embed maps into our activities, by means of a `MapView` widget.

And it worked.

More importantly, from the standpoint of users, the results from our apps were visually indistinguishable from the built-in Maps application available on devices that had the Maps SDK add-on.

This was the case through most of 2009. Eventually, though, the Google Maps team wanted to update the Maps application... but, for whatever reason, the decision was made to *not* update the Maps SDK add-on as well. At this point, the Google Maps team effectively forked the Maps SDK add-on, causing the Maps application to diverge from what other Android app developers could deliver. Over time, this feature gap became quite pronounced.

The release of Android 3.0 in early 2011 compounded the problems. Now, we needed to consider using fragments to help manage our code and deliver solutions to all screen sizes. Alas, while we could add maps to our fragments, we could only do so on API Level 11 or higher — the fragments backport from the Android Support package did not work with the Maps SDK add-on.

The release of Maps V2 helped all of this significantly. Now we have proper map support for native and backported versions of the fragment framework. We also have a look and feel that is closer to what the Maps application itself supports. While we still cannot reach feature parity with the Maps application, our SDK apps can at least look like they belong on the same device as the Maps application.

More importantly, as of the time of this writing, Maps V1 is no longer an option for new developers. Those who already have Maps V1 API keys can use Maps V1, but no new Maps V1 API keys are being offered. That leaves you with either using Maps V2 or some alternative mapping solution.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Where You Can Use Maps V2

Many devices will be able to use Maps V2... but not all. Notably:

- Devices need to support OpenGL ES 2.0+, to handle the new vector-based tiles that Maps V2 uses. Over 99% of Android devices in use today that support the Play Store (or its "Android Market" predecessor) also support OpenGL ES 2.0+.
- Devices will need an update to the Google Services Framework that accompanies the Play Store. Devices that do not have the Play Store — either because they are forever stuck on the old Android Market or, like the Kindle Fire, never had Play Store support in the first place — will be unable to use Maps V2.

Later in this chapter, we will look at other mapping libraries that you could use instead of either of Google's mapping solutions.

For many developers, the biggest limitation at the present time is that Maps V2 support does not exist in the Android emulator images. Google has indicated that this will be resolved in the not-too-distant future. While there are recipes online for hacking Maps V2 support into other emulator images, these recipes rely upon pirated versions of various pieces of Google software and therefore are not recommended.

Licensing Terms for Maps V2

As with the original Maps SDK add-on, to use Maps V2, you must agree to a terms of service agreement to be authorized to embed Google Maps within your application. If you intend to use Maps V2, you should review these terms closely, as they place many restrictions on developers. The most notorious of these is that you cannot use Maps V2 to create an application that offers "real time navigation or route guidance, including but not limited to turn-by-turn route guidance that is synchronized to the position of a user's sensor-enabled device."

If you find these terms to be an issue for your application, you may need to consider alternative mapping solutions.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What You Need to Start

If you wish to use Maps V2 in one or more of your Android applications, this section will outline what you need to get started.

Your Signing Key Fingerprint(s)

As with the legacy Maps SDK add-on, you will need fingerprints of your app signing keys, to tie your apps to your Google account and the API keys you will be generating. However, unlike the legacy Maps SDK add-on, the fingerprints you will be using will be created using the SHA–1 hash algorithm, rather than MD5.

First, you will need to know where the keystore is for your signing key. For a production keystore that you created yourself for your production apps, you should know where it is located already. For the debug keystore, used by default during development, the location is dependent upon operating system:

- OS X and Linux: `~/.android/debug.keystore`
- Windows XP: `C:\Documents and Settings\$USER\.android\debug.keystore`
- Windows Vista and Windows 7: `C:\Users\$USER\.android\debug.keystore`

(where `$USER` is your Windows user name)

You will then need to run the `keytool` command, to dump information related to this keystore. The `keytool` command is in your Java SDK, not the Android SDK. You will need to run this from a command line (e.g., Command Prompt in Windows). The specific command to run is:

```
keytool -list -v -keystore ... -alias androiddebugkey -storepass android
-keypass android
```

where the `...` is replaced by the path to your debug keystore, enclosed in quotation marks if the path contains spaces. For your production keystore, you would supply your own alias and passwords.

This should emit output akin to:

```
Alias name: androiddebugkey
Creation date: Aug 7, 2011
Entry type: PrivateKeyEntry
Certificate chain length: 1
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
Certificate[1]:
Owner: CN=Android Debug, O=Android, C=US
Issuer: CN=Android Debug, O=Android, C=US
Serial number: 4e3f2684
Valid from: Sun Aug 07 19:57:56 EDT 2011 until: Tue Jul 30 19:57:56 EDT 2041
Certificate fingerprints:
   MD5:  98:84:0E:36:F0:B3:48:9C:CD:13:EB:C6:D8:7F:F3:B1
   SHA1: E6:C5:81:EB:8A:F4:35:B0:04:84:3E:6E:C3:88:BD:B2:66:52:E7:09
   Signature algorithm name: SHA1withRSA
   Version: 3
```

You will need to make note of the `SHA1` entry (see third line from the bottom of the above sample).

Your Google Account

To sign up for an API key, you need a Google account. Ideally, this account would be the same one you intend to use for submitting apps to the Play Store (if, indeed, you intend to do so).

Your API Key

Given that you are logged into the aforementioned Google account, you can visit the Google Cloud Console to request access to the Maps V2 API. They have a tendency to keep changing this set of pages, but these instructions were good as of late February 2014:

- Create a project via the "Create project" option, if you have not done so already for something else (e.g., GCM)
- Open your project, then select "APIs & auth" from the left navigation bar, and in there select "APIs"
- Sift through the various APIs until you find "Google Maps Android API v2", then toggle that on
- Agree to the Terms of Service that appears when you try to toggle on Maps V2 access
- Click "Credentials" in the left navigation bar
- Click the "CREATE NEW KEY" button
- In the popup dialog, choose "Android key"
- In the fields that appear once you chose "Android key", fill in your app's package name and your SHA1 fingerprint, then click the "Create" button

This will give you an "API key" that you will need for your application.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you wish to have more than one app use Maps V2, you can click "Edit allowed Android applications" for a key, to return to the dialog where you can paste in another SHA1 fingerprint and package name, separated by a semicolon. Or, if you prefer, you can create new keys for each application.

For apps that are in (or going to) production, you will need to supply both the debug and production SHA1 fingerprints with your package name. By doing this on the same key, you will use the same API key string for both debug and production builds, which simplifies things a fair bit over the separate API keys you would have used with the legacy Maps SDK add-on.

Also note that a single API key seems to only support a few fingerprint/package pairs. If you try adding a new pair, and the form ignores you, you will need to set up a separate API key for additional pairs.

The Play Services Library

You also need to set up the Google Play Services library for use with your app.

First, you will need to download the "Google Play services" package in your SDK Manager (see highlighted line):

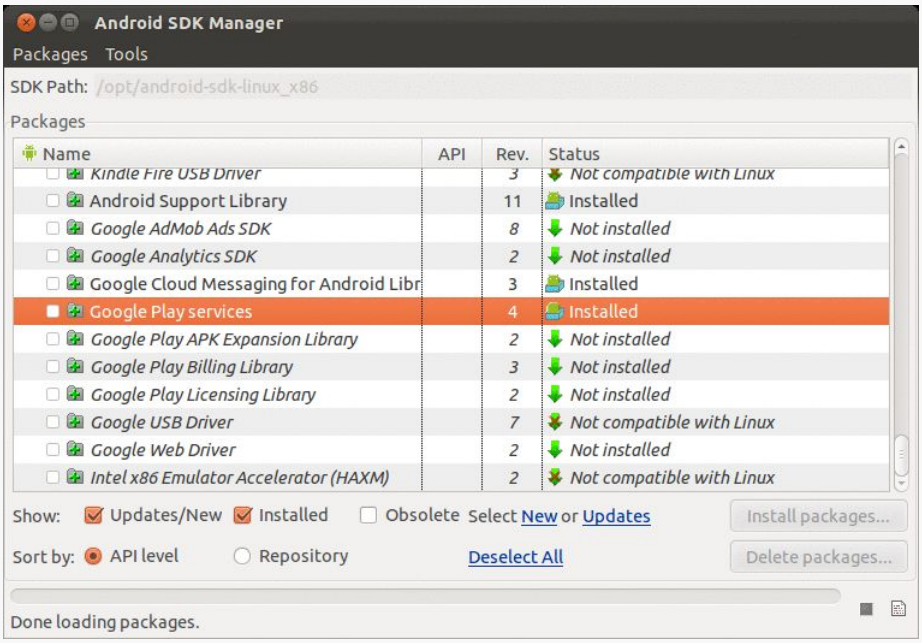


Figure 454: Android SDK Manager, Showing "Google Play services"

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio users will also want to download the "Google Repository", also in the same "Extras" area of the SDK Manager.

From there, things vary based on IDE.

Android Studio

Given that you have downloaded the above items, all you need to do is add a dependency on `com.google.android.gms:play-services-maps` for some likely version (e.g., `com.google.android.gms:play-services-maps:6.5.87`) to your `dependencies` closure.

Eclipse

You will need to add the Play Services Android library project to your workspace. You can do this via the "Import Existing Android Code into Workspace" wizard, pointing Eclipse to the `extras/google/google_play_services/libproject/google-play-services_lib/` directory inside of your Android SDK (or to a copy of it that you make elsewhere, if you prefer).

When you create a project that is to use Maps V2, you will need to add a reference to that library project, via the Eclipse project properties dialog.

Play Services and ProGuard

Note that the Play Services documentation requests that you add the following stanza to your `proguard-project.txt` file for use by your production builds:

```
-keep class * extends java.util.ListResourceBundle {
    protected Object[][] getContents();
}

-keep public class
com.google.android.gms.common.internal.safeparcel.SafeParcelable {
    public static final *** NULL;
}

-keepnames @com.google.android.gms.common.annotation.KeepName class *
-keepclassmembernames class * {
    @com.google.android.gms.common.annotation.KeepName *;
}

-keepnames class * implements android.os.Parcelable {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    public static final ** CREATOR;
}
```

The Book Samples… And You!

If you wish to try to run the book samples outlined in this chapter, you will need to make a few fixes to them for your own environment:

- Replace the Maps V2 API key in the manifest with your own
- Replace the reference to the Play Services Android library project with your own
- Change the build target to an Android SDK that you have downloaded (or download the Android SDK used by the project)

Setting Up a Basic Map

With that preparation work completed, now you can start working on projects that use the Maps V2 API. In this section, we will review the `MapsV2/Basic` sample project, which simply brings up a Maps V2 map on the world.

The Project Setup and the Manifest

This project uses Maps V2, and so it has a reference to that library project.

Our manifest file is fairly traditional, though there are a number of elements in it that are required by Maps V2:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.mapsv2.basic"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="16"/>

  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
  <uses-permission
android:name="com.google.android.providers.gsf.permission.READ_GSERVICES"/>
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>

  <uses-feature
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:glEsVersion="0x00020000"
    android:required="false"/>

  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Holo.Light.DarkActionBar">
    <activity
      android:name="MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>

    <meta-data
      android:name="com.google.android.maps.v2.API_KEY"
      android:value="AIzaSyC4iyT46cBOOIdKGcy5EmAxK5uCOQX2Oy8"/>

    <meta-data
      android:name="com.google.android.gms.version"
      android:value="@integer/google_play_services_version"/>

    <activity android:name="LegalNoticesActivity">
    </activity>
  </application>

</manifest>
```

Specifically:

- We need the `INTERNET`, `ACCESS_NETWORK_STATE`, and `WRITE_EXTERNAL_STORAGE` permissions, plus the `com.google.android.providers.gsf.permission.READ_GSERVICES` permission, all defined in `<uses-permission>` elements
- We need a `<meta-data>` element, with a name of `com.google.android.maps.v2.API_KEY`, whose value is the API key we got from the Google APIs Console for use with this particular package name
- We need a second `<meta-data>` element, with a name of `com.google.android.gms.version`, with a value of the `@integer/google_play_services_version` (an integer resource supplied by the Play Services SDK library project)

We also should include a `<uses-feature>` element for OpenGL ES 2.0. If your app absolutely must be able to run Maps V2, have `android:required="true"` (or drop

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the `android:required` attribute entirely, as `true` is the default), which will force devices to have OpenGL ES 2.0 to run your app. If your app will gracefully degrade for devices incapable of running Maps V2, use `android:required="false"`, as is shown in the sample.

Beyond those items, everything else in this project is based on what the app needs, more so than what Maps V2 needs.

Note that we used to need to define and use a custom permission, based upon our app's package name and ending in `MAPS_RECEIVE`. This is not required as of Play Services 3.1.59 and the "rev 8" release of the Play Services SDK.

The Play Services Detection

In the fullness of time, all devices that are capable of using Maps V2 will already have the on-device portion of this logic, known as the "Google Play services" app.

However, it is entirely possible, in the short term, that you will encounter devices that are capable of using Maps V2 (e.g., they have OpenGL ES 2.0 or higher), but do not have the "Google Play services" app from the Play Store, and therefore you cannot actually *use* Maps V2 in your app.

This is a departure from the Maps V1 approach, where either the device shipped with maps capability, or it did not, and nothing (legally) could be done to change that state.

To determine whether or not the Maps V2 API is available to you, the best option is to call the `isGooglePlayServicesAvailable()` static method on the `GooglePlayServicesUtil` utility class supplied by the Play Services library. This will return an `int`, with a value of `ConnectionResult.SUCCESS` if Maps V2 can be used right away.

Actually assisting the user to get Maps V2 set up all the way is conceivable but is also bug-riddled and annoying. The `MapsV2/Basic` sample app has an `AbstractMapActivity` base class that is designed to hide most of this annoyance from you. If you wish to know the details of how this works, we will cover it later in this chapter. The fact that this is bug-riddled also has impacts on whether or not you should be using Maps V2 at all for your app, a point which we will examine in an upcoming section.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Fragment and Activity

Our main activity — `MainActivity` — extends from the aforementioned `AbstractMapActivity` and simply overrides `onCreate()`, as most activities do:

```
package com.commonsware.android.mapsv2.basic;

import android.os.Bundle;

public class MainActivity extends AbstractMapActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (readyToGo()) {
      setContentView(R.layout.activity_main);
    }
  }
}
```

We call `setContentView()` to load up the `activity_main` layout resource:

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/map"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  class="com.google.android.gms.maps.MapFragment"/>
```

That resource, in turn, has a `<fragment>` element pointing to a `com.google.android.gms.maps.MapFragment` class supplied by the Play Services library. This is a fragment that knows how to display a Maps V2 map. There is a corresponding `com.google.android.gms.maps.SupportMapFragment` class for use with the fragments backport from the Android Support library.

You will notice, though, that we only call `setContentView()` if a `readyToGo()` method returns `true`. The `readyToGo()` method is supplied by the `AbstractMapActivity` class and returns `true` if we are safe to go ahead and use Maps V2, `false` otherwise. In the `false` case, `AbstractMapActivity` will be taking care of trying to get Maps V2 going, and we need do nothing further.

The License

According to the terms of use for Maps V2, you must show Maps V2 license information in your app's UI, in some likely spot. Apps that show their own license

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

terms, or have an "about" activity (or dialog) could display them there. Otherwise, you will need to have a dedicated spot for the Maps V2 license.

To obtain the license text, you can call `getOpenSourceSoftwareLicenseInfo()` on the `GooglePlayServicesUtil` utility class. This text can then be popped into a `TextView` somewhere in your app. `AbstractMapActivity` adds an action bar overflow item to display the license, which in turn invokes a `LegalNoticesActivity`, which simply displays the license text in a `TextView`. We will examine this in more detail later in this chapter.

The Result

When you run the app, assuming that Maps V2 is ready for use, you will get a basic map showing the west coast of Africa:

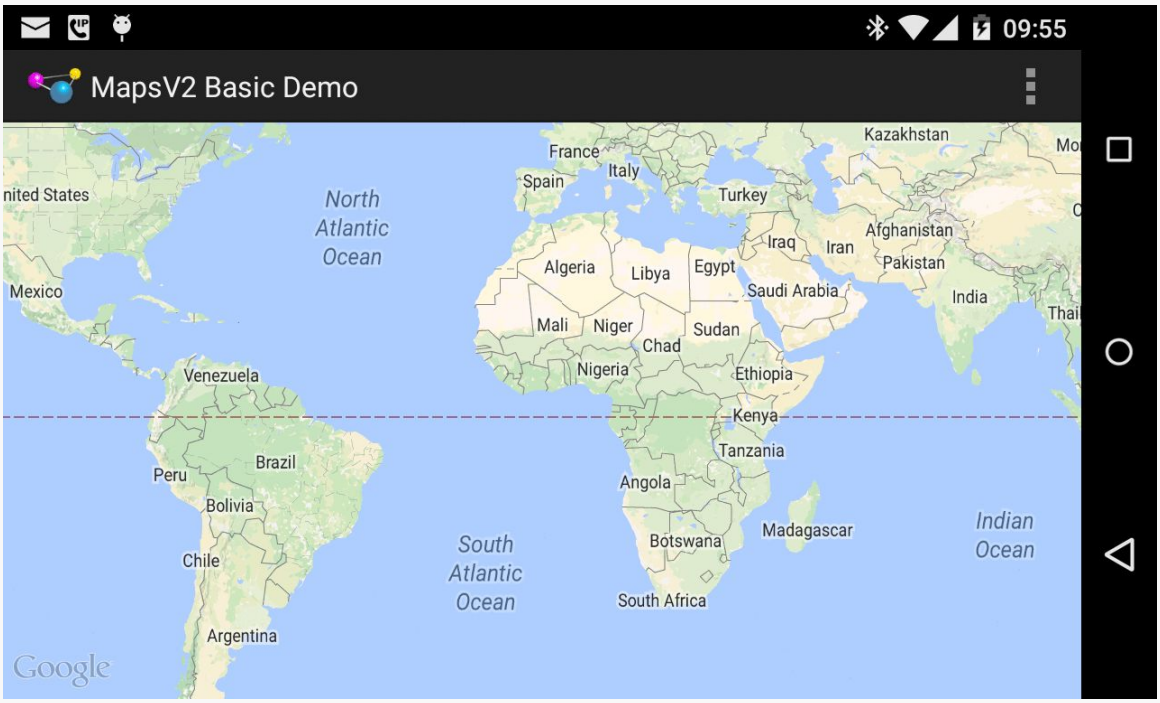


Figure 455: Maps V2 Map, as Initially Viewed

If you choose the "Legal Notices" action bar item, the view shifts to show a bunch of license terms:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

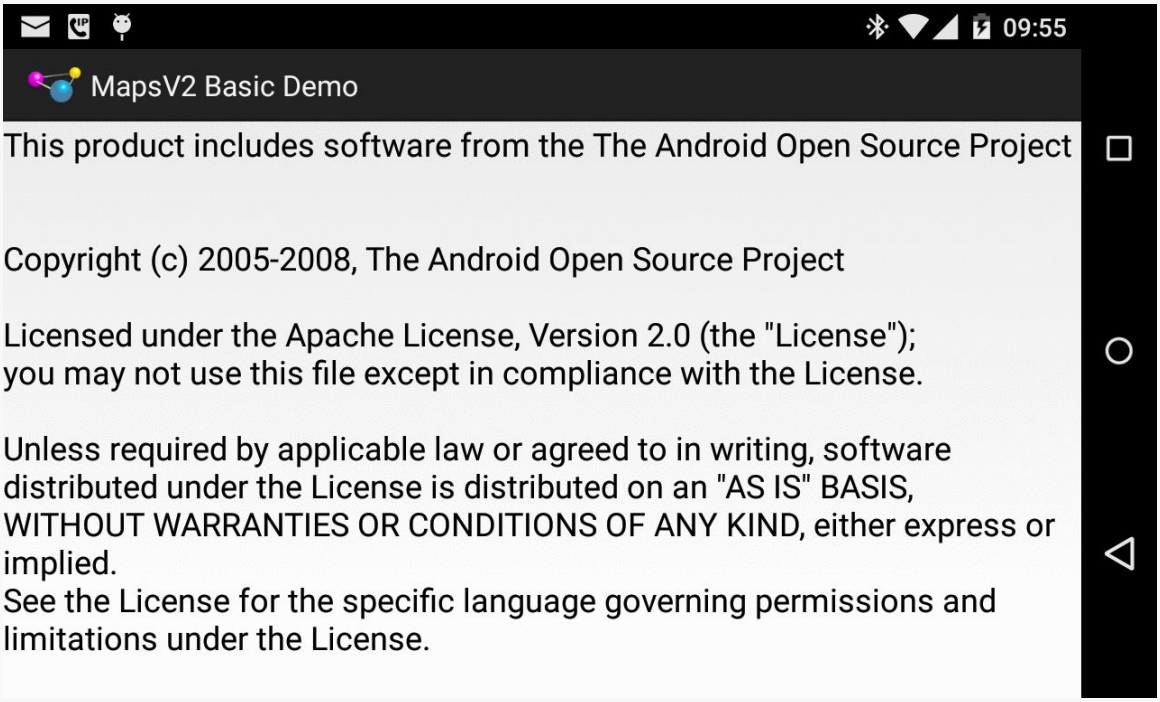


Figure 456: Maps V2 License Terms

If your Maps V2 API key is incorrect, or you do not have this app's package name set up for that key in the Google APIs Console, you will get an "Authorization failure" error message in LogCat, and you will get a blank map, akin to the behavior seen in Maps V1 when you had an invalid `android:apiKey` attribute on the `MapView`.

Playing with the Map

Showing a map of the west coast of Africa is nice, and it is entirely possible that is precisely what you wanted to show the user. If, on the other hand, you wanted to show the user something else — another location, a closer look, etc. — you will need to further configure your map, via a `GoogleMap` object.

To see how this is done, take a look at the `MapsV2/NooYawk` sample application. This is a clone of `MapsV2/Basic` that adds in logic to center and zoom the map over a portion of New York City.

The `onCreate()` method of the revised `MapActivity` is now a bit more involved:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onCreate(savedInstanceState);

    if (readyToGo()) {
      setContentView(R.layout.activity_main);

      MapFragment mapFrag=
          (MapFragment)getFragmentManager().findFragmentById(R.id.map);

      if (savedInstanceState == null) {
        mapFrag.getMapAsync(this);
      }
    }
  }
```

After calling `setContentView()`, we can retrieve our `MapFragment` via `findFragmentById()`, no different than any other static fragment.

Then, if `savedInstanceState` is `null` — meaning that the activity is not being recreated, but instead is being created from scratch — we call `getMapAsync()` on the `MapFragment`. This triggers some asynchronous work to set up a `GoogleMap` object. `getMapAsync()` takes an implementation of `OnMapReadyCallback` as a parameter. In this case, `OnMapReadyCallback` is implemented on the activity itself.

That `GoogleMap` object will then be delivered to us in `onMapReady()`. Most of our work in configuring the map will be accomplished by calling methods on this `GoogleMap` object.

To change where the map is centered, we can create a `CameraUpdate` object from the `CameraUpdateFactory` ("camera" in this case referring to the position of the user's virtual eyes with respect to the surface of the Earth). The `newLatLng()` factory method on `CameraUpdateFactory` will give us a `CameraUpdate` object that can re-center the map over a supplied latitude and longitude. Those coordinates are encapsulated in a `LatLng` object and are maintained as decimal degrees as Java `float` or `double` values (as opposed to the Maps V1 `GeoPoint`, which used integer microdegrees).

To change the zoom level of the map, we need another `CameraUpdate` object, this time from the `zoomTo()` factory method on `CameraUpdateFactory`. As with Maps V1, the zoom levels start at 1 and zoom in by powers of two. As you will see, a value of 15 gives you a nice block-level view of a city like New York City.

To actually apply these changes to the map, we have two methods on `GoogleMap`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. `moveCamera()` will perform a "smash cut" and immediately change the map based upon the supplied `CameraUpdate`
2. `animateCamera()` will smoothly animate the map from its original state to the new state supplied by the `CameraUpdate`

In our case, we immediately shift to the proper position, but then zoom in from the default zoom level to 15, giving us a map centered over Columbus Circle, in the southwest corner of Central Park in Manhattan:

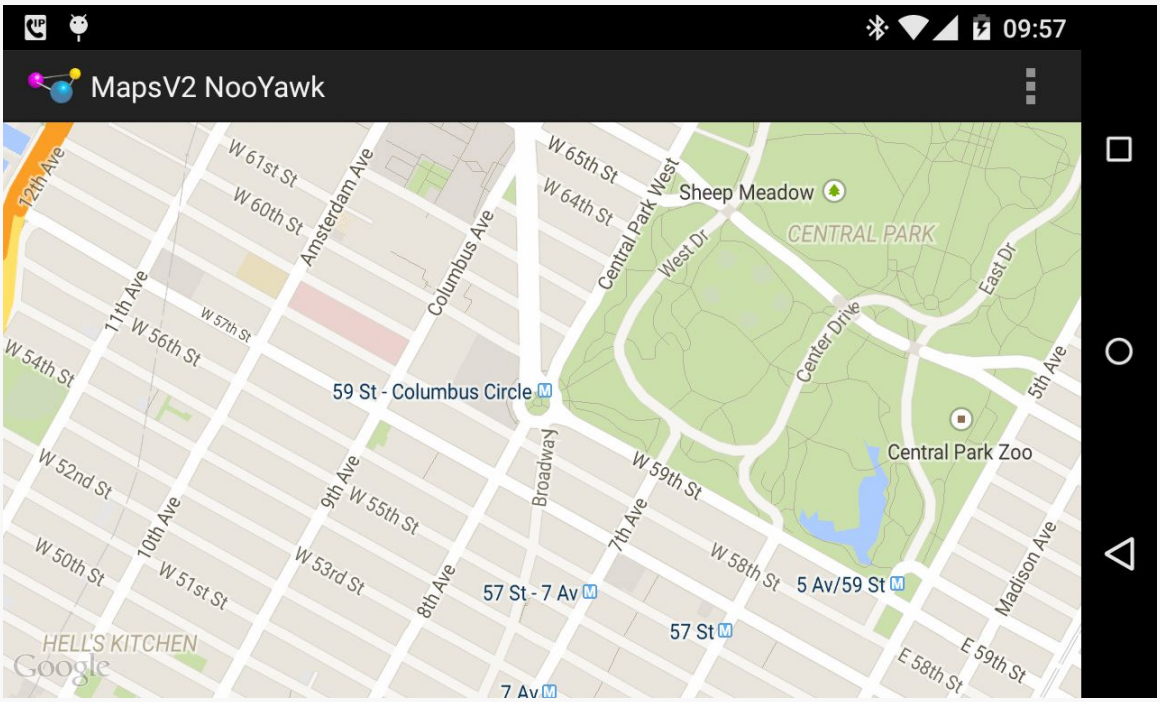


Figure 457: Maps V2 Centered Over Columbus Circle, New York City

Note that you might want to do both actions simultaneously, rather than have one be animated and one not as in this sample. In that case, you can manually create a `CameraPosition` object that describes the desired center, zoom, etc., then use the `newCameraPosition()` method on `CameraUpdateFactory` to get a `CameraUpdate` instance that will apply all of those changes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Map Tiles

The map, by default, shows the normal tile set. `setMapType()` on the `GoogleMap` allows you to switch to satellite, hybrid (satellite view plus place labels), or terrain tile sets.

Placing Simple Markers

In Maps V1, if we wanted to annotate a map, we used an `Overlay`. For markers — push-pins and the like — typically we would use an `ItemizedOverlay`, which handled a lot of the work of rendering these markers and responding to taps upon markers. However, `ItemizedOverlay` had its issues, particularly with regards to performance with large numbers of markers.

With Maps V2, the entire `Overlay` system is eliminated. Instead, you simply hand markers to the `GoogleMap` for display, as is illustrated in the `MapsV2/Markers` sample application. This is a clone of `MapsV2/NooYawk`, with four markers for four landmarks within Manhattan.

Our `onCreate()` method on `MainActivity` now always invokes `getMapAsync()`, not just when the activity is first created. However, we still check `savedInstanceState` and set a new `needsInit` `boolean` data member to `true` if `savedInstanceState` is `null`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  if (readyToGo()) {
    setContentView(R.layout.activity_main);

    MapFragment mapFrag=
        (MapFragment)getFragmentManager().findFragmentById(R.id.map);

    if (savedInstanceState == null) {
      needsInit=true;
    }

    mapFrag.getMapAsync(this);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Our `onMapReady()` method performs the camera adjustments if `needsInit` is `true`. It also has four additional statements – calls to a private `addMarker()` method to define the four landmarks:

```
@Override
public void onMapReady(GoogleMap map) {
  if (needsInit) {
    CameraUpdate center=
        CameraUpdateFactory.newLatLng(new LatLng(40.76793169992044,
                                                 -73.98180484771729));
    CameraUpdate zoom=CameraUpdateFactory.zoomTo(15);

    map.moveCamera(center);
    map.animateCamera(zoom);
  }

  addMarker(map, 40.748963847316034, -73.96807193756104,
            R.string.un, R.string.united_nations);
  addMarker(map, 40.76866299974387, -73.98268461227417,
      R.string.lincoln_center,
      R.string.lincoln_center_snippet);
  addMarker(map, 40.765136435316755, -73.97989511489868,
      R.string.carnegie_hall, R.string.practice_x3);
  addMarker(map, 40.70686417491799, -74.01572942733765,
      R.string.downtown_club, R.string.heisman_trophy);
}
```

The `addMarker()` method on our `MainActivity` adds markers by creating a `MarkerOptions` object and passing it to the `addMarker()` on `GoogleMap`. `MarkerOptions` offers a so-called "fluent" interface, with a series of methods to affect one aspect of the `MarkerOptions`, each of which returns the `MarkerOptions` object itself. That way, configuring a `MarkerOptions` is a chained series of method calls:

```
private void addMarker(GoogleMap map, double lat, double lon,
                       int title, int snippet) {
  map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                   .title(getString(title))
                                   .snippet(getString(snippet)));
}
```

Here, we:

- Set the `position()` of the marker, in the form of another `LatLng` object
- Set the `title()` and `snippet()` of the marker to be a pair of strings loaded from string resources

We will see other methods available on `MarkerOptions` in upcoming sections of this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`addMarker()` on `GoogleMap` returns an actual `Marker` object, which we could hold onto to change certain aspects of it later on (e.g., its title). In the case of this sample, we ignore this.

Now, you may be wondering why we set up the markers on every `onMapReady()` invocation, not just in the `needsInit` block. That is because while a `MapFragment` retains its camera information (center, zoom, etc.), it does *not* retain its markers on a configuration change. Hence, we need to re-establish the markers in all calls to `onCreate()`, not just the very first one.

With no other changes, we get a version of the map that shows markers at our designated locations:

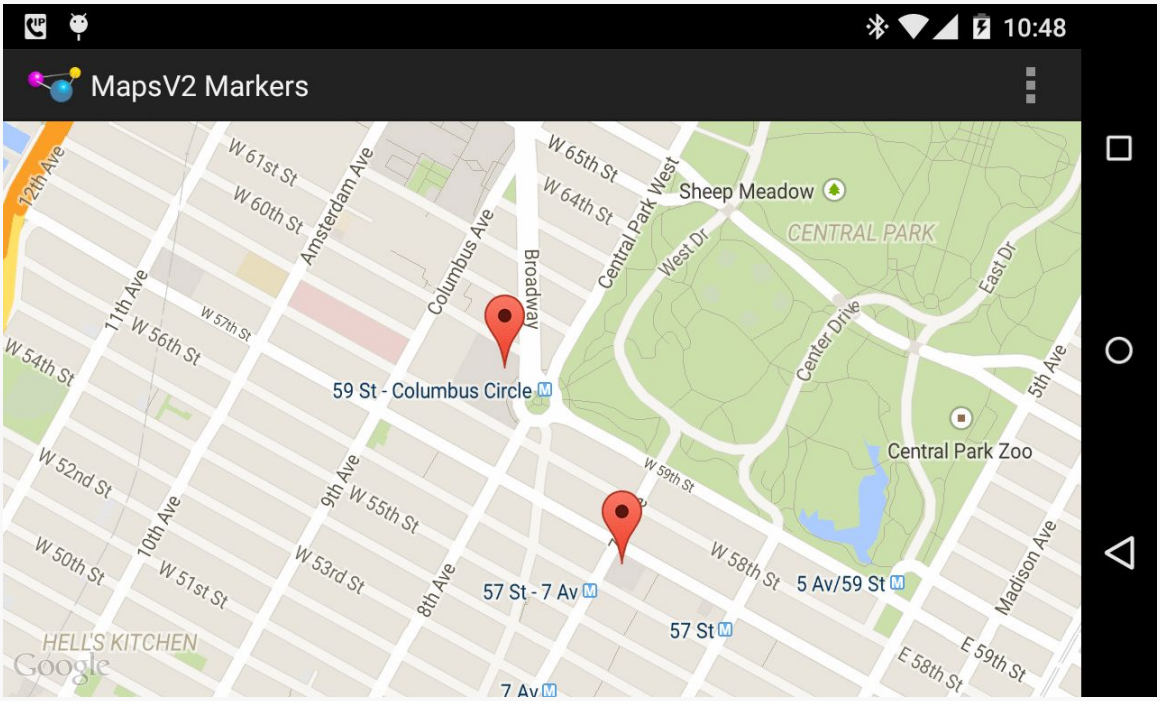


Figure 458: Maps V2 with Two Markers

Initially, we only see two markers, as the other two are outside the current center position and zoom level of the map. If the user changes the center or zoom, markers will come and go as needed:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

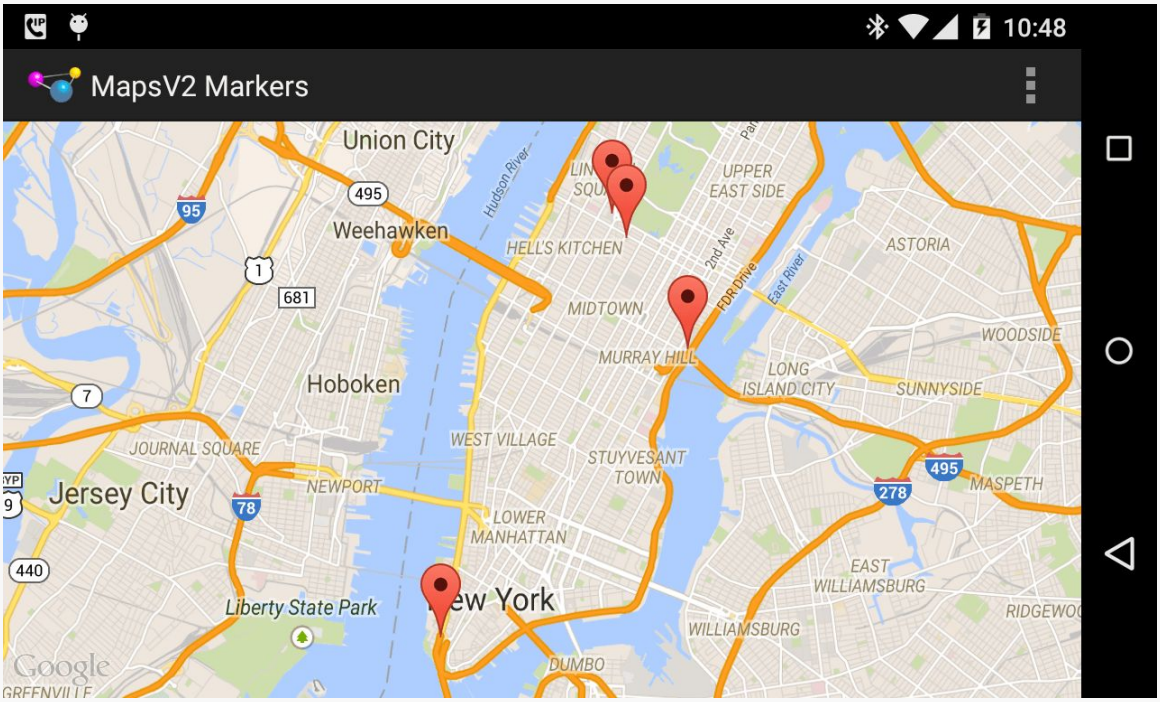


Figure 459: Maps V2 with All Four Markers

We do not need to worry about managing the markers ourselves, so long as the `GoogleMap` performance is adequate. It is likely that dumping 10,000 markers into a `GoogleMap` will still result in sluggish responses, though, so you may need to add and remove markers yourself based upon what portion of the world the user happens to be examining in the map at the moment.

Seeing All the Markers

When you add markers to a map, there is no guarantee that the markers will be visible given the map's current center position and zoom level. In fact, it is entirely possible that you add a bunch of markers and *none* are visible, so the user may not realize that the markers were added.

There is a way that you can center and zoom the map to show some set of markers, based on their positions. You get to choose the markers: all of them, the four nearest markers, etc.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We can see how this works in the MapsV2/Bounds sample application. This is a clone of MapsV2/Markers from the previous section, with reworked code to show all four markers when the map is first displayed.

The key to making this work is a LatLngBounds object. This represents a bounding box that contains all LatLng locations handed to the LatLngBounds. To build up a LatLngBounds, you can use the LatLngBounds.Builder class. So, our revised MainActivity has a LatLngBounds.Builder private data member:

```
  private LatLngBounds.Builder builder=new LatLngBounds.Builder();
```

Our revised addMarker() method adds the LatLng values from our markers as they are added to the map:

```
  private void addMarker(GoogleMap map, double lat, double lon,
                         int title, int snippet) {
    Marker marker=
        map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                         .title(getString(title))
                                         .snippet(getString(snippet)));

    builder.include(marker.getPosition());
  }
```

Finally, the revised onMapReady() moves the CameraUpdateFactory work until after all four of the addMarker() calls and changes it a bit:

```
  @Override
  public void onMapReady(final GoogleMap map) {
    addMarker(map, 40.748963847316034, -73.96807193756104,
              R.string.un, R.string.united_nations);
    addMarker(map, 40.76866299974387, -73.98268461227417,
              R.string.lincoln_center,
              R.string.lincoln_center_snippet);
    addMarker(map, 40.765136435316755, -73.97989511489868,
              R.string.carnegie_hall, R.string.practice_x3);
    addMarker(map, 40.70686417491799, -74.01572942733765,
              R.string.downtown_club, R.string.heisman_trophy);

    if (needsInit) {
      findViewById(android.R.id.content).post(new Runnable() {
        @Override
        public void run() {
          CameraUpdate allTheThings=
              CameraUpdateFactory.newLatLngBounds(builder.build(), 32);

          map.moveCamera(allTheThings);
        }
      });
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }
```

Specifically, we:

- Ask the `LatLngBounds.Builder` to `build()` the `LatLngBounds`
- Pass that to the `newLatLngBounds()` method on `CameraUpdateFactory`, along with an inset value in pixels (all `LatLng` locations will be that many pixels in from the edges, or more)
- Use `moveCamera()` to center and zoom the map based upon the resulting `CameraUpdate`

All of this is done in a `Runnable` which we `post()` to a `View` (here, the `FrameLayout` of our activity supplied by Android as `android.R.id.content`). `GoogleMap` cannot ensure that all of our markers are visible until it knows how big the map is, and that is not known until the map is rendered to the screen. `post()` will add our work to the end of the main application thread's work queue. The `Runnable` will not be run until after the map is on the screen, at which time the `CameraUpdate` can work.

Flattening and Rotating Markers

Markers, by default, appear to be "push pins" pressed into the surface of the map. This is not necessarily obvious with the default top-down perspective of the map camera. But, if you use a two-finger vertical swiping gesture, you can change the camera tilt, and that will illustrate the "push pin" effect a bit better:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

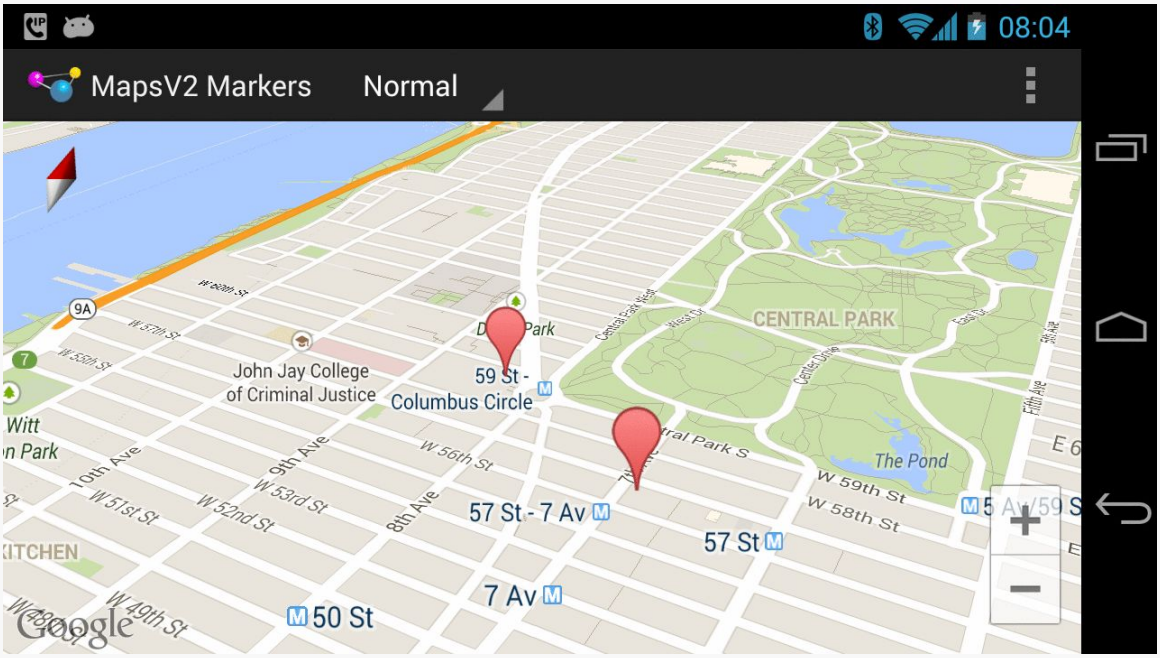


Figure 460: Maps V2 with Markers, Viewed on a Tilt

However, you have options for flat markers and rotated markers.

A flat marker is one that is flat on the map. In other words, rather than theoretically rising out of the Z axis of the map, the marker is kept on the X-Y plane:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

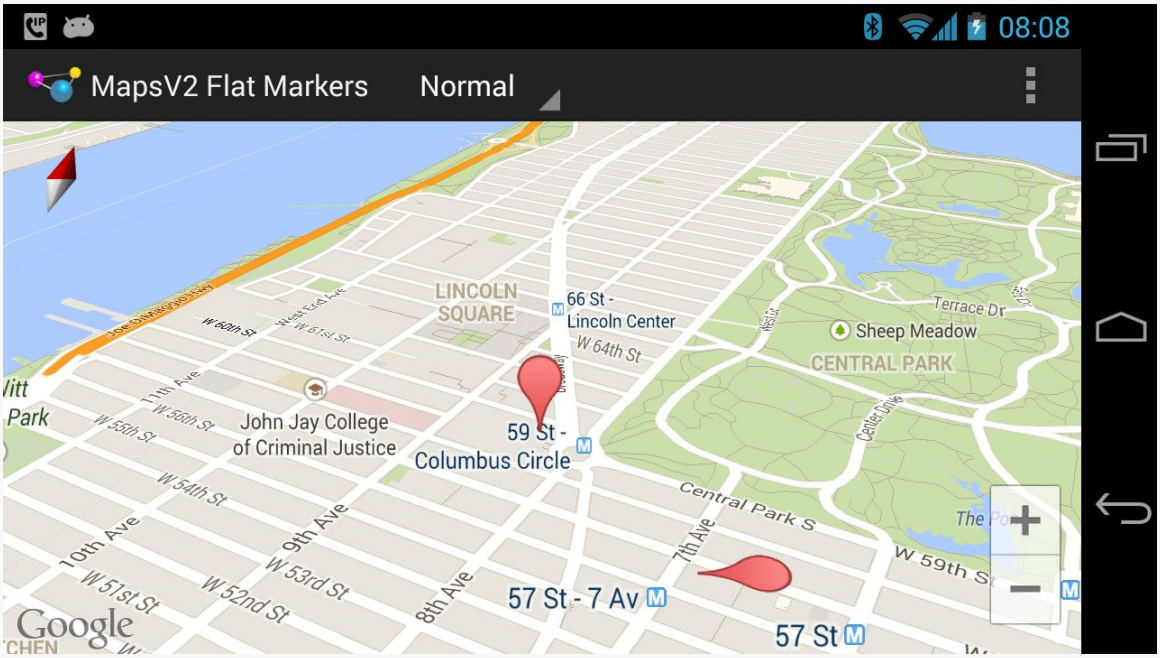


Figure 461: Maps V2 with Markers, One Normal, One Flat

It is also possible to rotate a marker. The flat marker in the previous screenshot is rotated 90 degrees from its normal "bulb on the north side" orientation. The following screenshot shows another flat marker, rotated 270 degrees from normal:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

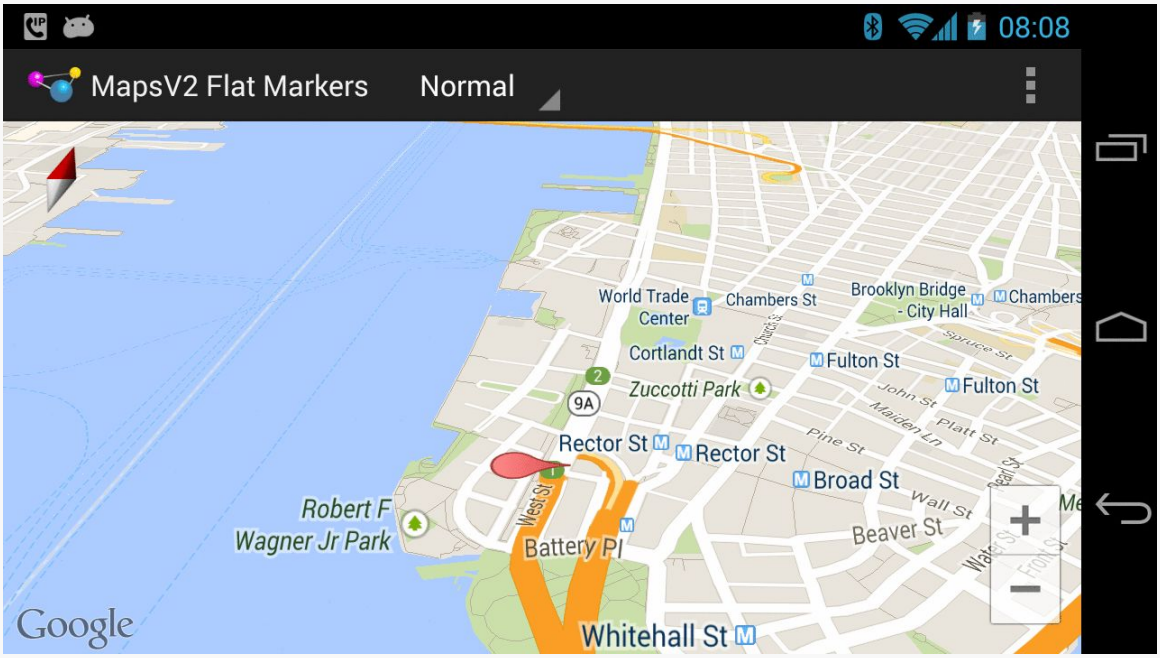


Figure 462: Maps V2 with Markers, Flat and Rotated

These features can be handy for providing pointers in a particular direction, such as indicating not only the location to make a turn, but what direction to turn at that location.

These capabilities are courtesy of new `flat()` and `rotation()` methods on `MarkerOptions`, plus corresponding getters and setters on `Marker` itself. To see how this works, let's examine the `MapsV2/FlatMarkers` sample application. This is a clone of `MapsV2/Markers`, with markers applied using different values for `flat()` and `rotation()`.

Specifically, our own `addMarker()` helper method now takes and applies a `boolean` parameter for `flat` (`true` means it is flat, `false` means normal behavior), as well as a `float` parameter for `rotation` (a value between `0` and `360` for the rotation off the default in degrees):

```
private void addMarker(GoogleMap map, double lat, double lon,
                       int title, int snippet, boolean flat,
                       float rotation) {
  map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                   .title(getString(title))
                                   .snippet(getString(snippet))
                                   .flat(flat).rotation(rotation));
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

When we call `addMarker()`, we supply corresponding values:

```
    addMarker(map, 40.748963847316034, -73.96807193756104,
              R.string.un, R.string.united_nations, false, 180);
    addMarker(map, 40.76866299974387, -73.98268461227417,
              R.string.lincoln_center,
              R.string.lincoln_center_snippet, false, 0);
    addMarker(map, 40.765136435316755, -73.97989511489868,
              R.string.carnegie_hall, R.string.practice_x3, true, 90);
    addMarker(map, 40.70686417491799, -74.01572942733765,
              R.string.downtown_club, R.string.heisman_trophy, true,
              270);
```

Sprucing Up Your “Info Windows”

If the user taps on one of the markers from the preceding sample, Android will automatically display a popup, known as an “info window”:

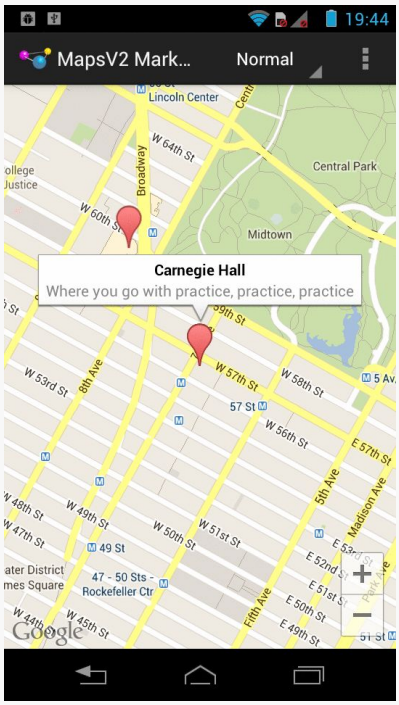


Figure 463: Maps V2 with Default Info Window

You can tailor that “info window” if desired, either replacing just the interior portion (leaving the bounding border with its caret intact) or replacing the entire window. However, in the interests of memory conservation, you do not hand new `View`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

widgets to the `MarkerOptions` object. Instead, you can provide an adapter that will be called when info windows (or their contents) are required.

To see how this works, we can examine the `MapsV2/Popups` sample application. This is a clone of `MapsV2/Markers`, where we are using our own layout file for the contents of the info windows, from the `popup.xml` layout resource:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal">

  <ImageView
    android:id="@+id/icon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:padding="2dip"
    android:src="@drawable/ic_launcher"
    android:contentDescription="@string/icon"/>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
      android:id="@+id/title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="25sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/snippet"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="15sp"/>
  </LinearLayout>

</LinearLayout>
```

Here, we will show the title and snippet in our own chosen font size and weight, plus show the launcher icon on the left.

To use this layout, we must create an `InfoWindowAdapter` implementation — in the case of this sample project, that is found in the `PopupAdapter` class:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.mapsv2.popups;

import android.annotation.SuppressLint;
import android.view.LayoutInflater;
import android.view.View;
import android.widget.TextView;
import com.google.android.gms.maps.GoogleMap.InfoWindowAdapter;
import com.google.android.gms.maps.model.Marker;

class PopupAdapter implements InfoWindowAdapter {
  private View popup=null;
  private LayoutInflater inflater=null;

  PopupAdapter(LayoutInflater inflater) {
    this.inflater=inflater;
  }

  @Override
  public View getInfoWindow(Marker marker) {
    return(null);
  }

  @SuppressLint("InflateParams")
  @Override
  public View getInfoContents(Marker marker) {
    if (popup == null) {
      popup=inflater.inflate(R.layout.popup, null);
    }

    TextView tv=(TextView)popup.findViewById(R.id.title);

    tv.setText(marker.getTitle());
    tv=(TextView)popup.findViewById(R.id.snippet);
    tv.setText(marker.getSnippet());

    return(popup);
  }
}
```

When an info window is to be displayed, Android will first call `getInfoWindow()` on our `InfoWindowAdapter`, passing in the `Marker` whose info window is needed. If we return a `View` here, that will be used for the entire info window. If, instead, we return `null`, Android will call `getInfoContents()`, passing in the same `Marker` object. If we return a `View` here, Android will use that as the "body" of the info window, with Android supplying the border. If we return `null`, the default info window is displayed. This way, we can conditionally do any of the three possibilities (replace the window, replace the contents, or accept the default).

In our case, `getInfoContents()` will inflate the `popup.xml` layout and populate the two `TextView` widgets with the title and snippet from the `Marker`. However, we cache

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the inflated layout and reuse it on the second and subsequent calls to `getInfoContents()`. Despite the "adapter" name conjuring up visions of `ListAdapter` and having multiple outstanding views, `InfoWindowAdapter` will only ever use one `View` at a time. Hence, rather than inflate our layout each time we need to show the info window, we can safely reuse the previously-used `View`.

Then, we just need to tell the `GoogleMap` to use our `InfoWindowAdapter`, via a call to `setInfoWindowAdapter()`, such as this statement from `onMapReady()` of our new edition of `MainActivity`:

```
    map.setInfoWindowAdapter(new PopupAdapter(getLayoutInflater()));
```

Now, when the user taps on a marker, they will get our customized info window:

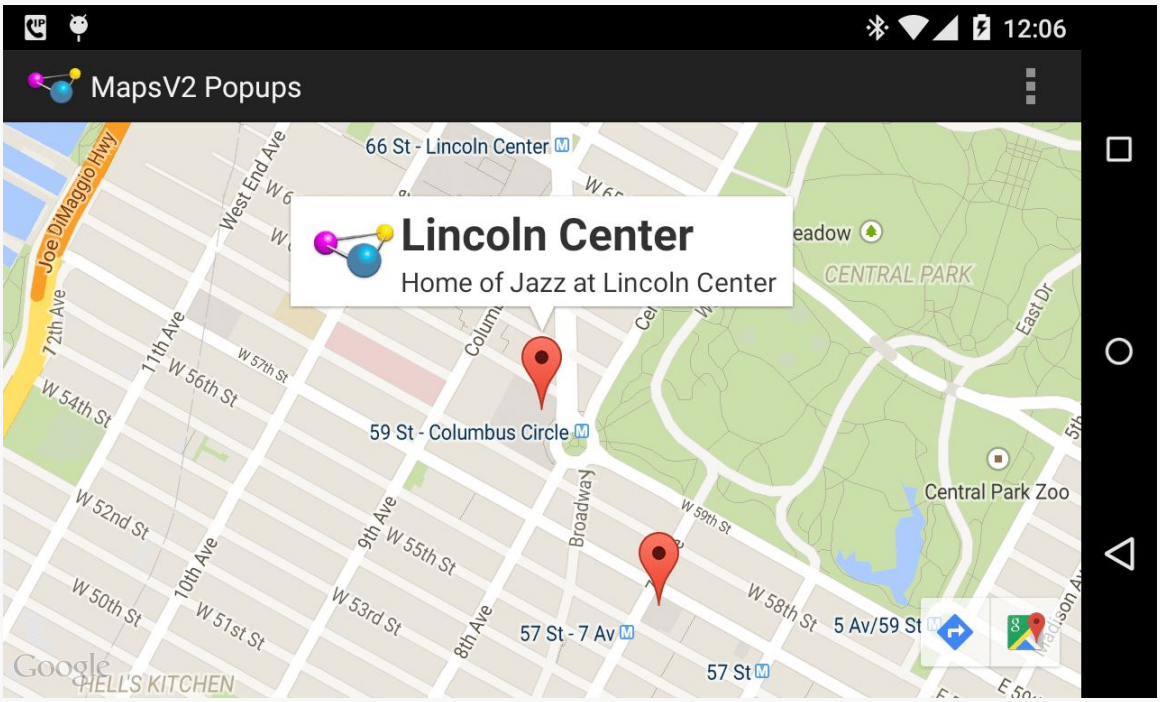


Figure 464: Maps V2 with Customized Info Window

We can also call `setOnInfoWindowClickListener()` on our `GoogleMap`, passing in an implementation of the `OnInfoWindowClickListener` interface, to find out when the user taps on the info window. In the case of `MainActivity`, we set up the activity itself to implement that interface and be the listener:

```
    map.setOnInfoWindowClickListener(this);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This requires us to implement an `onInfoWindowClick()` method, where we are passed the `Marker` representing the tapped-upon info window:

```
@Override
public void onInfoWindowClick(Marker marker) {
  Toast.makeText(this, marker.getTitle(), Toast.LENGTH_LONG).show();
}
```

Here, we just display a `Toast` with the title of the `Marker` when the user taps an info window:

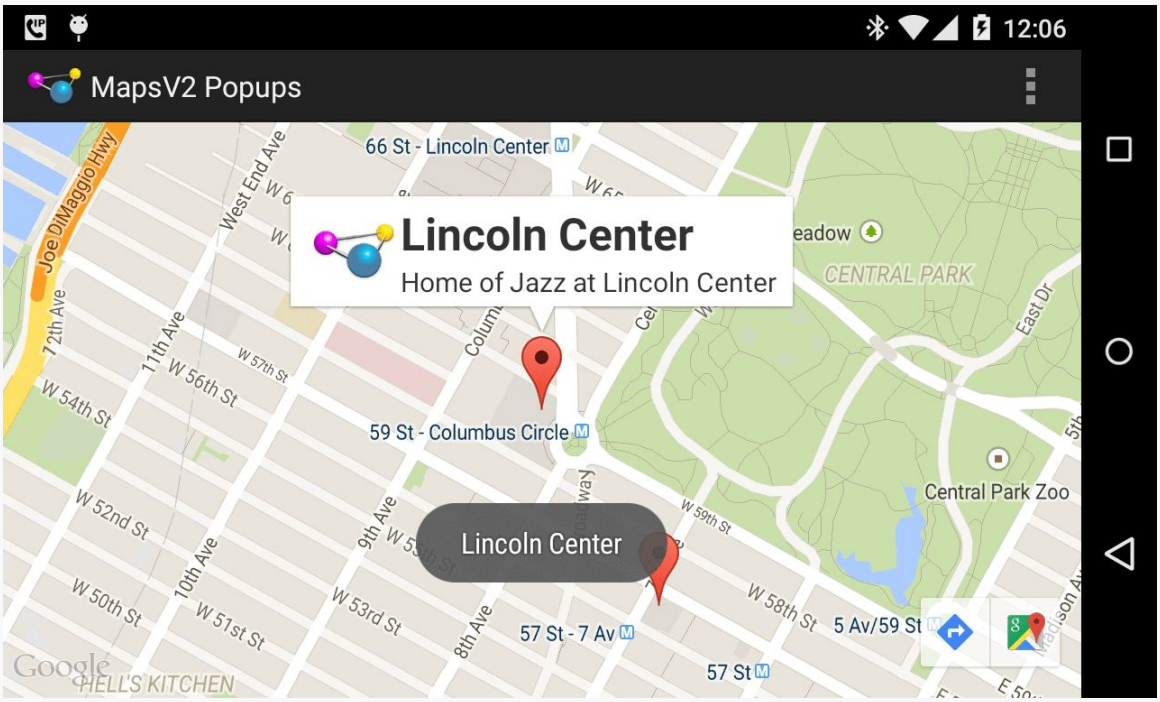


Figure 465: Maps V2 with Toast Triggered by Tap on Info Window

Note that, according to the documentation, you can only find out about taps on the entire info window. Indeed, if you try setting up click listeners on the widgets in your custom layout, you will find that they are not called. This is because the `View` you return for the info window is converted into a `Bitmap`, which is then displayed. Presumably, this is to steer developers in the direction of making larger tap targets, rather than expecting users to tap tiny elements within an info window. On the other hand, if your design calls for a large info window containing several navigation options, you will need to either re-think your design or avoid the info window system. We will see how to find out about taps on markers more directly later in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Images and Your Info Window

The `Bitmap` approach that Maps V2 uses for the info window introduces an additional challenge: updating the info window itself. Normally, we would just update the individual widgets in the info window, the way we might update widgets in an already-visible row in a `ListView`. However, that is not an option here, as our widgets are discarded almost immediately.

One particular occurrence of this problem comes when you want to show an image in the info window. If the image is a resource, or is already in memory, showing it is not a big problem, as you can just populate your `ImageView` in your info window with it. However, if the image is a file (or, worse, needs to be downloaded), you want to load the image asynchronously. However, if you kick off some background thread, like an `AsyncTask`, to retrieve the image, you will return from your `InfoWindowAdapter` method long before the task is complete. Your info window will show whatever placeholder image you used; the image you loaded will never be seen, even if you update your original `ImageView`.

There are two solutions to this problem.

The best solution, by far, is to have the images before you need them, wherever possible. For example, if you are showing a map with 25 markers, for which you need 25 thumbnail images, start downloading those images while you are showing the map. With luck, at the point in time when the user taps on a marker to show the info window, you will have your image already.

However, this approach will not work well if:

- You need a ridiculous number of images, or
- You need images, but they need to be downloaded full-sized and turned into thumbnails locally, as that might consume quite a bit of bandwidth, or
- Your last name is Murphy, and therefore the user taps on an info window before you have had a chance to prepare its image

The workaround is to make note of the `Marker` the user tapped upon to open its info window, then call `showInfoWindow()` on that `Marker` to cause the info window to be redisplayed once you have your image, triggering calls to your `InfoWindowAdapter`. There, you can see that your image cache includes the image that you need, and you can apply it to the info window.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The problem *here* is that it is possible that the user tapped on another marker, after the first one, while you were busily fetching and loading the image. Hence, rather than blindly calling `showInfoWindow()` on the `Marker`, you should call `isInfoWindowShown()` first, and only call `showInfoWindow()` to force the refresh if `isInfoWindowShown()` returns `true`. Otherwise, some other marker's info window is shown. The user is not expecting this earlier info window to somehow magically reappear.

All of this is a pain. It can be made a bit less of a pain by use of an image fetching-and-caching library like Picasso. We can see how this can be applied by looking at the `MapsV2/ImagePopups` sample application. This is a clone of `MapsV2/Popups`, with some additions to handle lazy-populating an info window based upon a downloaded image.

First, since we are going to be generating some thumbnails based on downloaded imagery, it helps to establish a fixed-size `ImageView` for our icon. So, this project has a pair of dimension resources, for the image height and width:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="icon_width">96dp</dimen>
    <dimen name="icon_height">64dp</dimen>

</resources>
```

Those are then used in a revised version of the popup layout resource:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal">

  <ImageView
    android:id="@+id/icon"
    android:layout_width="@dimen/icon_width"
    android:layout_height="@dimen/icon_height"
    android:padding="2dip"
    android:src="@drawable/ic_launcher"
    android:contentDescription="@string/icon"/>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:orientation="vertical">

    <TextView
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      android:id="@+id/title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="25sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/snippet"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="15sp"/>
  </LinearLayout>

</LinearLayout>
```

We need some way of keeping track of what images should be used for each marker. This is somewhat annoying to implement, as we cannot subclass `Marker`, since it is marked as `final` and cannot be extended. However, we can use `getId()` on a `Marker` to obtain a unique ID, and we can use that as the key to additional model data. We will examine variations on this technique later in this chapter. For now, this sample gets away with a simple `HashMap`, mapping the string ID of a `Marker` to a `Uri` representing an image to be shown for that `Marker`'s info window:

```
  private HashMap<String, Uri> images=new HashMap<String, Uri>();
```

Our private `addMarker()` method now takes a `String` name of an image, and it adds a `Uri` pointing to that image to the `HashMap`, keyed by the ID of the generated `Marker`:

```
  private void addMarker(GoogleMap map, double lat, double lon,
                         int title, int snippet, String image) {
    Marker marker=
        map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                         .title(getString(title))
                                         .snippet(getString(snippet)));

    if (image != null) {
      images.put(marker.getId(),
                 Uri.parse("http://misc.commonsware.com/mapsv2/"
                     + image));
    }
  }
```

For three of our markers, we pass in actual filenames; for a fourth, `null` is used, indicating that there is no suitable image for use:

```
    addMarker(map, 40.748963847316034, -73.96807193756104,
              R.string.un, R.string.united_nations, "UN_HQ.jpg");
    addMarker(map, 40.76866299974387, -73.98268461227417,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
              R.string.lincoln_center,
              R.string.lincoln_center_snippet,
              "Avery_Fisher_Hall.jpg");
    addMarker(map, 40.765136435316755, -73.97989511489868,
              R.string.carnegie_hall, R.string.practice_x3,
              "Carnegie_Hall.jpg");
    addMarker(map, 40.70686417491799, -74.01572942733765,
              R.string.downtown_club, R.string.heisman_trophy, null);
```

Note that the three images being used in this chapter come from Wikipedia. One is public domain, the others are licensed under the Creative Commons Attribution 1.0 license. Those two are a picture of Avery Fisher Hall, part of the Lincoln Center for the Performing Arts (courtesy of Geographer) and the other is a picture of the United Nations building (courtesy of WorldIslandInfo).

The `PopupAdapter` needs access to these images. It will also need access to a `Context`, for use with Picasso. So, `PopupAdapter` now has data members for these, which are passed into a revised version of its constructor by `MainActivity`. That constructor not only holds onto the new objects, but it retrieves the values of the dimension resources for our images, converted by Android into pixels for the screen density of the device that we are running on:

```
  PopupAdapter(Context ctxt, LayoutInflater inflater,
               HashMap<String, Uri> images) {
    this.ctxt=ctxt;
    this.inflater=inflater;
    this.images=images;

    iconWidth=
        ctxt.getResources().getDimensionPixelSize(R.dimen.icon_width);
    iconHeight=
        ctxt.getResources().getDimensionPixelSize(R.dimen.icon_height);
  }
```

The revised `getInfoContents()` method is significantly more complicated than was its predecessor:

```
  @SuppressLint("InflateParams")
  @Override
  public View getInfoContents(Marker marker) {
    if (popup == null) {
      popup=inflater.inflate(R.layout.popup, null);
    }

    if (lastMarker == null
        || !lastMarker.getId().equals(marker.getId())) {
      lastMarker=marker;

      TextView tv=(TextView)popup.findViewById(R.id.title);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      tv.setText(marker.getTitle());
      tv=(TextView)popup.findViewById(R.id.snippet);
      tv.setText(marker.getSnippet());

      Uri image=images.get(marker.getId());
      ImageView icon=(ImageView)popup.findViewById(R.id.icon);

      if (image == null) {
        icon.setVisibility(View.GONE);
      }
      else {
        Picasso.with(ctxt).load(image).resize(iconWidth, iconHeight)
               .centerCrop().noFade()
               .placeholder(R.drawable.placeholder)
               .into(icon, new MarkerCallback(marker));
      }
    }

    return(popup);
  }
```

We track the last `Marker` that we have processed in a `lastMarker` data member. Initially, of course, that will be `null`. If it is, or if the `Marker` passed into `getInfoContents()` is a different one (based on the `getId()` value), then we populate the popup `View`. This includes fetching the `Uri` from the `HashMap` of `Uri` values (given the `Marker` ID). If there is no `Uri`, `getInfoContents()` marks the `ImageView` as `GONE`, so it will not take up space in the popup. If, however, there *is* an image `Uri`, `getInfoContents()` asks Picasso to "do its thing":

- Load the image from the `Uri`
- Resize the image to be the desired dimensions for the `ImageView`, center-cropping to keep the right aspect ratio
- Skip the fade-in animation that is normally applied when Picasso populates an `ImageView` (as the Maps V2 `Bitmap` is generated before the animation completes, resulting in a washed-out image)
- Use a particular `placeholder` drawable resource while the image is loading
- Populate the `ImageView` with the results, specifying a `MarkerCallback` to be notified of the results

`MarkerCallback`, as an implementation of Picasso's `Callback` interface, needs `onError()` and `onSuccess()` methods. `onError()` just dumps a message to LogCat, while `onSuccess()` refreshes the info window, via a call to `showInfoWindow()` on the `Marker`, if that info window is still showing:

```
  static class MarkerCallback implements Callback {
    Marker marker=null;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  MarkerCallback(Marker marker) {
    this.marker=marker;
  }

  @Override
  public void onError() {
    Log.e(getClass().getSimpleName(), "Error loading thumbnail!");
  }

  @Override
  public void onSuccess() {
    if (marker != null && marker.isInfoWindowShown()) {
      marker.showInfoWindow();
    }
  }
}
```

If you run this sample app, you will see the popup with a placeholder image at first, quickly being replaced by the thumbnail supplied by Picasso:

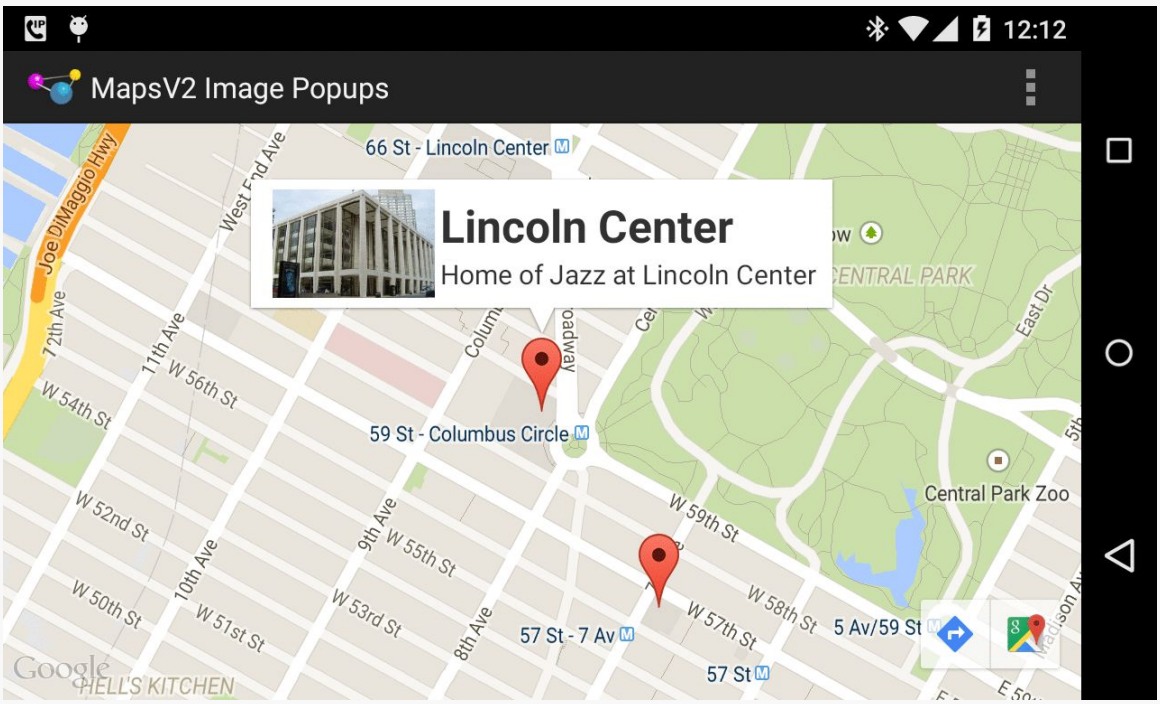


Figure 466: Maps V2 with Popup and Thumbnail

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Setting the Marker Icon

Maps V2 includes a stock marker icon that looks a lot like the standard Google Maps marker. You have three major choices for what to use for your own markers:

1. Stick with the stock icon, which is the default behavior
2. Change the stock icon to a different hue
3. Replace the stock icon with your own from an asset, resource, file, or in-memory `Bitmap`

To indicate that you want a different icon than the stock one, use the `icon()` method on the `MarkerOptions` fluent interface. This takes a `BitmapDescriptor`, which you get from one of a series of static methods on the `BitmapDescriptorFactory` class.

For example, you might have a revised version of the `addMarker()` method of `MainActivity` that took a hue — a value from 0 to 360 representing different colors along a color wheel. 0 represents red, 120 represents green, and 240 represents blue, with different shades in between. There is a series of `HUE_` constants defined on `BitmapDescriptorFactory`, plus a `defaultMarker()` method that takes a hue as a parameter and returns a `BitmapDescriptor` that will use the stock icon, colored to the specified hue:

```
  private void addMarker(GoogleMap map, double lat, double lon,
                         int title, int snippet, int hue) {
    map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                     .title(getString(title))
                                     .snippet(getString(snippet))

.icon(BitmapDescriptorFactory.defaultMarker(hue)));
  }
```

This could then be used to give you different colors per marker, or by category of marker, etc.:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

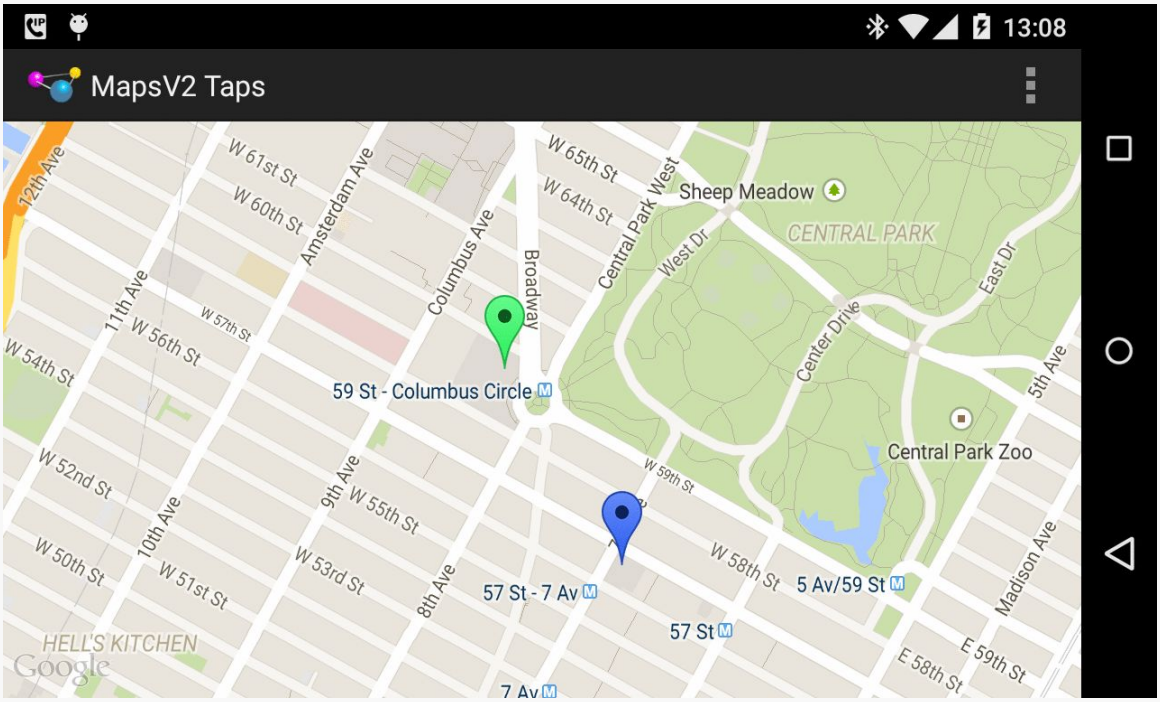


Figure 467: Maps V2 with Alternate Marker Hues

Note that you can modify the icon at runtime via the `setIcon()` method on the `Marker` returned by `addMarker()` method on `GoogleMap`.

However, you cannot draw the marker directly yourself, the way you might have with Maps V1. What you *can* do is draw to a `Bitmap`-backed `Canvas` object, then use the resulting `Bitmap` with `BitmapFactoryDescriptor` and its `fromBitmap()` factory method.

Responding to Taps

Perhaps we would like to find out when a user taps on one of our markers, instead of displaying an info window. Maybe we want to have some other UI response to that tap in our app.

To do that, simply create an implementation of the `OnMarkerClickListener` interface and attach it to the `GoogleMap` via `setOnMarkerClickListener()`. You will then be called with `onMarkerClick()` when the user taps on a marker, and you are passed the `Marker` object in question. If you return `true`, you are indicating that you

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

are handling the event; returning `false` means that default handling (the info window) should be done.

You can see this, plus the multi-colored markers, in the `MapsV2/Taps` sample application. This takes `MapsV2/Popups` and adds a `Toast` when the user taps a marker, in addition to displaying the info window:

```
@Override
public boolean onMarkerClick(Marker marker) {
  Toast.makeText(this, marker.getTitle(), Toast.LENGTH_LONG).show();

  return(false);
}
```

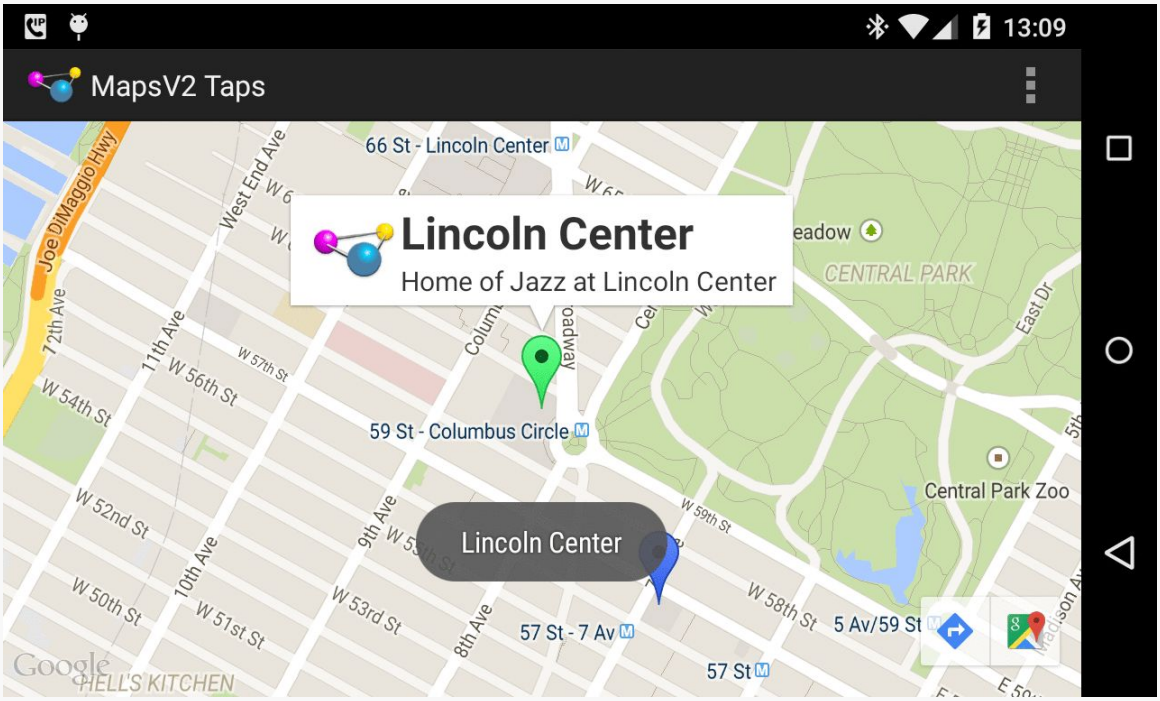


Figure 468: Maps V2 with Toast and Info Window

Our call to `setOnMarkerClickListener()` is up in the `onMapReady()` method of `MainActivity`:

```
    map.setOnMarkerClickListener(this);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Dragging Markers

In Maps V1, if you wanted to drag-and-drop map markers, you had to handle all of that yourself, using low-level touch events.

Conversely, Maps V2 handles marker drag-and-drop for you.

By default, markers are not draggable. But, if you call `draggable(true)` on your `MarkerOptions` when creating the marker — or call `setDraggable(true)` on the `Marker` later on — Android will automatically support drag-and-drop. The user can tap-and-hold on the marker to enable drag mode, then slide the marker around the map.

Note that at the present time, this functionality is a little odd. When you tap-and-hold the marker, with drag mode enabled, the marker initially jumps away from its original position. The user can reposition the marker to any desired location, and the marker will seem to "drop" where the user requests. Why the marker makes the sudden shift at the outset, using the default marker settings, is unclear.

Of course, your code may need to know about drag-and-drop events, such as to update your own data model to reflect the newly-chosen location. You can register an `OnMarkerDragListener` that will be notified of the start of the drag, where the marker slides during the drag, and where the marker is dropped at the end of the drag.

You can see all of this in the `MapsV2/Drag` sample application, which is a clone of `MapsV2/Popup` with drag-and-drop enabled.

To enable drag-and-drop, we just chain `draggable(true)` onto the series of calls on our `MarkerOptions` when creating the markers:

```
  private void addMarker(GoogleMap map, double lat, double lon,
                         int title, int snippet) {
    map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                     .title(getString(title))
                                     .snippet(getString(snippet))
                                     .draggable(true));
  }
```

We also register `MainActivity` as being the drag listener, up in `onMapReady()`:

```
    map.setOnMarkerDragListener(this);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That requires `MainActivity` to implement `OnMarkerDragListener`, which in turn requires three methods to be defined: `onMarkerDragStart()`, `onMarkerDrag()`, and `onMarkerDragEnd()`:

```
@Override
public void onMarkerDragStart(Marker marker) {
  LatLng position=marker.getPosition();

  Log.d(getClass().getSimpleName(), String.format("Drag from %f:%f",
                                                  position.latitude,
                                                  position.longitude));
}

@Override
public void onMarkerDrag(Marker marker) {
  LatLng position=marker.getPosition();

  Log.d(getClass().getSimpleName(),
        String.format("Dragging to %f:%f", position.latitude,
                      position.longitude));
}

@Override
public void onMarkerDragEnd(Marker marker) {
  LatLng position=marker.getPosition();

  Log.d(getClass().getSimpleName(), String.format("Dragged to %f:%f",
      position.latitude,
      position.longitude));
}
```

Here, we just dump the information about the new marker position in LogCat.

So, if you run this app and drag-and-drop a marker, you will see output in LogCat akin to:

```
12-19 13:10:36.442: D/MainActivity(22510): Drag from 40.770876:-73.982499
12-19 13:10:36.892: D/MainActivity(22510): Dragging to 40.770876:-73.981593
12-19 13:10:36.912: D/MainActivity(22510): Dragging to 40.770795:-73.981352
12-19 13:10:36.932: D/MainActivity(22510): Dragging to 40.770754:-73.981141
.
.
.
12-19 13:10:38.292: D/MainActivity(22510): Dragging to 40.769596:-73.983615
12-19 13:10:38.372: D/MainActivity(22510): Dragged to 40.769596:-73.983615
```

The actual list of events was much longer, as `onMarkerDrag()` is called a *lot*, so the . . . in the LogCat entries above reflect another 50 or so lines for a drag-and-drop that took a couple of seconds.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Also, up in `onCreate()`, we retain our `MapFragment` across configuration changes via `setRetainInstance(true)`:

```
      mapFrag.setRetainInstance(true);
```

Retaining the fragment instance causes the fragment to keep our markers in their moved positions, rather than resetting them to their original positions.

The "Final" Limitations

With Maps V1 and `ItemizedOverlay`, `OverlayItem` was a regular Java class. You could extend this class, to support additional data or behaviors that were unique to your app.

Of course, an even more flexible approach would have been for `OverlayItem` to be an interface, with a stock `SimpleOverlayItem` concrete implementation. That way, you could implement `OverlayItem` on some Java class resident elsewhere in your app's class hierarchy (e.g., inheriting from some class needed for a persistence framework).

Rather than taking a step forward, Maps V2 took a step backward instead.

In Maps V2, not only do you not create `Marker` objects directly yourself, but `Marker` is marked as `final` and cannot be extended.

However, `Marker` does have `getId()`, an immutable identifier for the `Marker`. We can use that as a key for a `HashMap` that allows us to get at additional model data associated with the `Marker`.

You can see this approach in the `MapsV2/Models` sample application, which is a clone of `MapsV2/Popup` where we use the ID in just this fashion.

Our simplified model is merely the data we poured into our `Marker` objects in the original `MapsV2/Popup` project:

```
package com.commonsware.android.mapsv2.model;

import android.content.Context;

public class Model {
  String title;
  String snippet;
  double lat;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  double lon;

  Model(Context ctxt, double lat, double lon, int title,
        int snippet) {
    this.title=ctxt.getString(title);
    this.snippet=ctxt.getString(snippet);
    this.lat=lat;
    this.lon=lon;
  }

  String getTitle() {
    return(title);
  }

  String getSnippet() {
    return(snippet);
  }

  double getLatitude() {
    return(lat);
  }

  double getLongitude() {
    return(lon);
  }
}
```

Our activity holds onto a `HashMap` of these `Model` objects, with the map keyed by the `Marker` ID (a `String`):

```
  private HashMap<String, Model> models=new HashMap<String, Model>();
```

Of course, a real application would have a much more elaborate setup than this.

We then arrange to populate our map with `Marker` objects created from our `Model` objects, moving the add-the-markers-to-the-map logic to an `addMarkers()` method:

```
  private void addMarkers(GoogleMap map) {
    Model model=
        new Model(this, 40.748963847316034, -73.96807193756104,
                  R.string.un, R.string.united_nations);

    models.put(addMarkerForModel(map, model).getId(), model);

    model=
        new Model(this, 40.76866299974387, -73.98268461227417,
                  R.string.lincoln_center,
                  R.string.lincoln_center_snippet);
    models.put(addMarkerForModel(map, model).getId(), model);

    model=
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        new Model(this, 40.765136435316755, -73.97989511489868,
                  R.string.carnegie_hall, R.string.practice_x3);
    models.put(addMarkerForModel(map, model).getId(), model);

    model=
        new Model(this, 40.70686417491799, -74.01572942733765,
                  R.string.downtown_club, R.string.heisman_trophy);
    models.put(addMarkerForModel(map, model).getId(), model);
  }

  private Marker addMarkerForModel(GoogleMap map, Model model) {
    LatLng position=
        new LatLng(model.getLatitude(), model.getLongitude());

    return(map.addMarker(new MarkerOptions().position(position)
                                            .title(model.getTitle())
                                            .snippet(model.getSnippet())));

  }
```

Notice that `addMarkerForModel()` returns the `Marker`, and we use `getId()` on that `Marker` as the key when adding a `Model` to the `HashMap`.

Our `PopupAdapter` gets the data for the info window from the `Model` (though, in truth, in this case, it could have gotten the data from the `Marker` itself, since we did not add more information to the info window):

```
package com.commonsware.android.mapsv2.model;

import android.view.LayoutInflater;
import android.view.View;
import android.widget.TextView;
import java.util.HashMap;
import com.google.android.gms.maps.GoogleMap.InfoWindowAdapter;
import com.google.android.gms.maps.model.Marker;

class PopupAdapter implements InfoWindowAdapter {
  LayoutInflater inflater=null;
  HashMap<String, Model> models=null;

  PopupAdapter(LayoutInflater inflater, HashMap<String, Model> models) {
    this.inflater=inflater;
    this.models=models;
  }

  @Override
  public View getInfoWindow(Marker marker) {
    return(null);
  }

  @Override
  public View getInfoContents(Marker marker) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    View popup=inflater.inflate(R.layout.popup, null);

    TextView tv=(TextView)popup.findViewById(R.id.title);

    tv.setText(models.get(marker.getId()).getTitle());
    tv=(TextView)popup.findViewById(R.id.snippet);
    tv.setText(models.get(marker.getId()).getSnippet());

    return(popup);
  }
}
```

Visually, this is indistinguishable from the original `MapsV2/Popups` project. Of course, a real app would have more complex models, perhaps containing more discrete information for a more complex info window.

A Bit More About IPC

IPC is not only a problem in terms of disappearing `Marker` objects.

If you run a Maps V2 app under Traceview, to see what methods get called and how much time everything takes, you will see that many, many operations with `GoogleMap` do little in your process, but instead make synchronous calls to a Play Services process to do the real work:

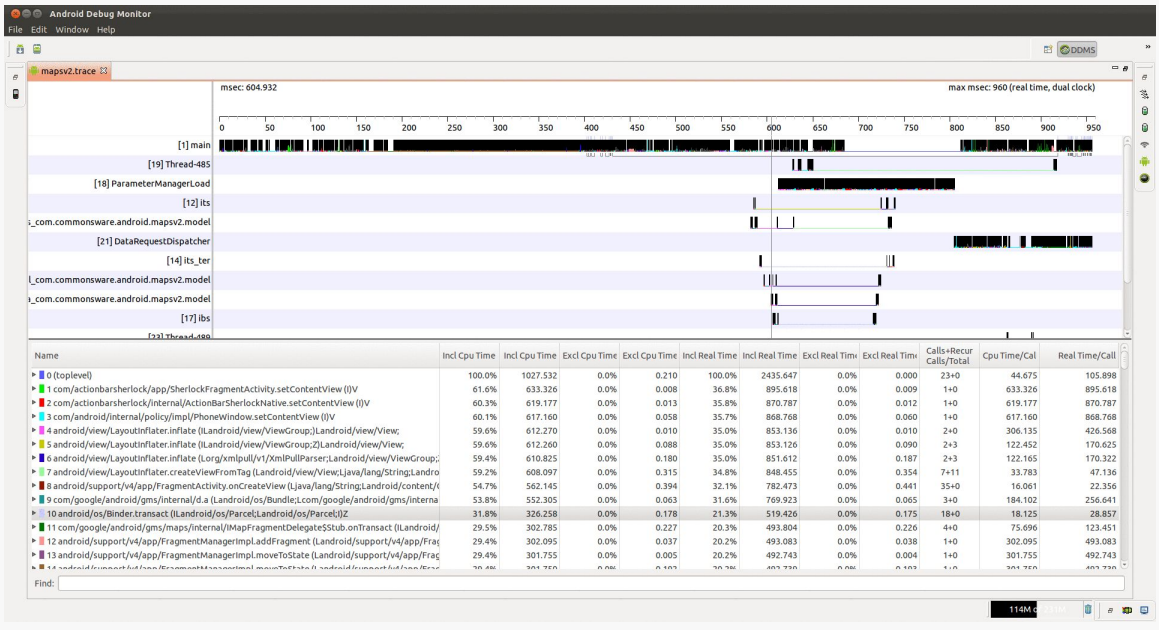


Figure 469: Traceview Results for Maps V2 Map Creation

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The preceding trace came from just the `onCreate()` method of the `MapsV2/Models` sample from the preceding section. Over 30% of the time to run `onCreate()` is tied up in IPC calls. And, unfortunately, you are not allowed to do much manipulation of a `GoogleMap` from a background thread (e.g., `moveCamera()`).

The moral of this story is to avoid manipulating your `GoogleMap` in time-sensitive portions of your code.

(the author would once again like to thank Cyril Mottier for pointing out this limitation in Maps V2)

Finding the User

Many times, the user is looking at a map to figure out where they are. Perhaps they are lost. Perhaps their spouse or significant other thinks that they are lost. Perhaps they think that they were teleported somewhere (e.g., a North African desert) after turning a "frozen wheel" in an icy cavern beneath an island, and therefore are *really* lost. Stranger things have happened.

(well, OK, perhaps not)

Regardless, it is often useful to help point out to the user their current location. That is a matter of adding a suitable location permission (e.g., `ACCESS_FINE_LOCATION`) and calling `setMyLocationEnabled(true)` on your `GoogleMap`. This activates a layer that will highlight their location, with the user having an option of having the "camera" (i.e., their perspective on the map) reposition itself to their location and move as they move. This latter capability is activated by a small icon in the upper right of the map.

You can see this in operation in the `MapsV2/MyLocation` sample application, which is a clone of `MapsV2/Popup` with standard location tracking enabled.

All we do is call two additional methods on our `GoogleMap` in `onCreate()`:

- `setMyLocationEnabled()`, indicating that we want the "my location" layer added and automatic tracking to be enabled, and
- `setOnMyLocationChangeListener()`, indicating that we *also* want to be notified about changes in the user position

```
    map.setMyLocationEnabled(true);
    map.setOnMyLocationChangeListener(this);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That latter method also requires our activity to implement the `OnMyLocationChangeListener` interface, which in turn requires us to implement the `onMyLocationChange()` method, which will be called when Maps V2 gets a new location fix:

```
@Override
public void onMyLocationChange(Location lastKnownLocation) {
  Log.d(getClass().getSimpleName(),
      String.format("%f:%f", lastKnownLocation.getLatitude(),
          lastKnownLocation.getLongitude()));
}
```

Here, we simply log the location to LogCat.

This is nice and easy, giving us our my-location overlay and arrow indicating the user's location and orientation:

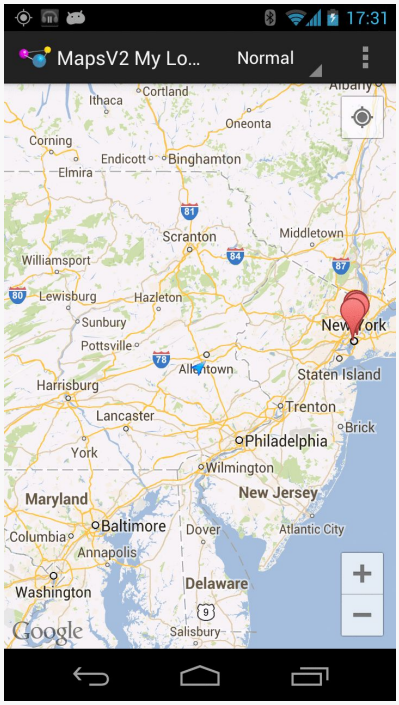


Figure 470: Maps V2, Showing the User's Location

However, there are two problems here. First, `setOnMyLocationChangeListener()` is now deprecated, as Google would prefer that you directly request the locations through the `LocationClient` available from Play Services.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Second, there does not appear to be a way to *force* camera tracking of the user's position — you are reliant upon the user tapping that icon. You also have no control over the nature of the location provider that is used.

However, there is a workaround for this, proposed in a Stack Overflow answer – provide your own location data and update the camera yourself, by means of `setLocationSource()`. `setLocationSource()` lets *you* push locations to the `GoogleMap`, making other adjustments (e.g., camera position) along the way.

To see how this works, take a peek at the `MapsV2/Location` sample application, which is a clone of `MapsV2/Popup` with custom location tracking enabled.

Along with adding `ACCESS_FINE_LOCATION` to the manifest, this sample project adds some lines to the `onMapReady()` implementation of `MainActivity` to configure the `GoogleMap`:

```
    locMgr=(LocationManager)getSystemService(LOCATION_SERVICE);
    crit.setAccuracy(Criteria.ACCURACY_FINE);
    locMgr.requestLocationUpdates(0L, 0.0f, crit, this, null);

    map.setLocationSource(this);
    map.setMyLocationEnabled(true);
    map.getUiSettings().setMyLocationButtonEnabled(false);
```

The first three lines get access to a `LocationManager`, indicate that a `Criteria` object (initialized as a data member) should require fine accuracy, and request location updates. These come from the location tracking subsystem in Android.

The next two lines register our activity as being the source of location data and turns on location tracking in the `GoogleMap`, so the user's position will be marked on the map.

The last line disables the user's control over whether the camera position tracks their movement, since we want that to always be on in this case.

In `onResume()` and `onPause()` of `MainActivity`, we enable and disable getting location updates, as is typical of an activity needing location data. However, we also tell the `GoogleMap` that we are going to supply it with location data, rather than it having to obtain location data itself:

```
  @Override
  public void onResume() {
    super.onResume();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (locMgr!=null) {
      locMgr.requestLocationUpdates(0L, 0.0f, crit, this, null);
    }

    if (map!=null) {
      map.setLocationSource(this);
    }
  }

  @Override
  public void onPause() {
    map.setLocationSource(null);
    locMgr.removeUpdates(this);

    super.onPause();
  }
```

Note that we are blindly assuming that we will get location data. A production-grade app would put in better smarts to confirm that we will actually learn our location via this `Criteria` (e.g., the user does not have all location providers disabled).

Also note that because the map initialization is now happening on a background thread, `onResume()` might be called before `onMapReady()`, and so `onResume()` has to check to see if we already have a `LocationManager` and `GoogleMap` before proceeding.

The call to `setLocationSource()` — both in `onMapReady()` and `onResume()` – tells `GoogleMap` that our `MainActivity` itself is to be the source of location data. This requires `MainActivity` to implement the `LocationSource` interface, requiring us to implement `activate()` and `deactivate()` methods:

```
  @Override
  public void activate(OnLocationChangedListener listener) {
    this.mapLocationListener=listener;
  }

  @Override
  public void deactivate() {
    this.mapLocationListener=null;
  }
```

`activate()` provides us with an `OnLocationChangedListener`, from `GoogleMap`, to which we need to pass location data as we get it. `deactivate()` indicates that we should no longer attempt to contact that listener. In addition to holding onto that listener (or removing our reference to it when deactivated), we also take this opportunity to request and remove location updates.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `onLocationChanged()` method — where we get our location fixes from `LocationManager` via the `LocationListener` interface — must pass the location along to the `GoogleMap`-supplied `OnLocationChangedListener`, if we have such a listener available:

```
@Override
public void onLocationChanged(Location location) {
  if (mapLocationListener != null) {
    mapLocationListener.onLocationChanged(location);

    LatLng latlng=
        new LatLng(location.getLatitude(), location.getLongitude());
    CameraUpdate cu=CameraUpdateFactory.newLatLng(latlng);

    map.animateCamera(cu);
  }
}
```

Here, we also create a `CameraUpdate` representing the new location and animate that update, to have the map slide over to the new location, centering the camera on the user's updated position.

The net effect of all of this is that the map continuously re-centers itself to show the user's position, which `GoogleMap` is highlighting on the map for us.

Drawing Lines and Areas

If you wanted to draw on a map in the Maps V1 framework, you created an `Overlay` and drew upon it. This forced you to handle low-level drawing work yourself, as you were handed a `Canvas` object and had to handle all the lines, fills, and so forth yourself.

Maps V2 offers a different approach. Free-form drawing is still conceivable, though it appears to have to be handled in the form of tile overlays instead of map overlays. However, for the simpler cases of drawing lines and areas, Maps V2 has built-in polyline, polygon, and circle support. You tell the `GoogleMap` what needs to be drawn, and it handles drawing it, both initially and as the map is zoomed or panned. A polyline is a line connecting a series of points; a polygon is a region defined by a series of corners. A circle, from the standpoint of Maps V2, is defined by a center coordinate and a radius.

We can see polylines and polygons on a `GoogleMap` in the `MapsV2/Poly` sample application, which is a clone of `MapsV2/Popup` with two additions:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- A polyline connecting the locations of our four markers
- A polygon enclosing the area of Manhattan known as the Garment District (bounded by 34th Street, 42nd Street, Fifth Avenue, and Ninth Avenue)

To draw those, we simply add a few lines to `onMapReady()` of `MainActivity`:

```
    PolylineOptions line=
        new PolylineOptions().add(new LatLng(40.70686417491799,
                                             -74.01572942733765),
                                  new LatLng(40.76866299974387,
                                             -73.98268461227417),
                                  new LatLng(40.765136435316755,
                                             -73.97989511489868),
                                  new LatLng(40.748963847316034,
                                             -73.96807193756104))
                             .width(5).color(Color.RED);

    map.addPolyline(line);

    PolygonOptions area=
        new PolygonOptions().add(new LatLng(40.748429, -73.984573),
                                 new LatLng(40.753393, -73.996311),
                                 new LatLng(40.758393, -73.992705),
                                 new LatLng(40.753484, -73.980882))
                            .strokeColor(Color.BLUE);

    map.addPolygon(area);
```

The API for adding polylines and polygons is reminiscent of the API for adding markers: define an `...Options` object with the characteristics of the item to be drawn, then call an `add...()` method on `GoogleMap` to add the item.

So, to add a polyline, we create a `PolylineOptions` object. Using its fluent interface, we `add()` a series of `LatLng` objects, representing the points to be connected by the line. We also specify the line width in *pixels* via `width()` and the color of the line via `color()`. If we had several lines that might overlap, we could specify the `zIndex()`, where higher indexes indicate lines to be drawn over the top of lines with lower indexes. We add the polyline to the map by passing our `PolylineOptions` to `addPolyline()` on `GoogleMap`.

This gives us a line connecting the four markers, with `GoogleMap` handling the details of where the line should be drawn on the screen given the current map center and zoom levels:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

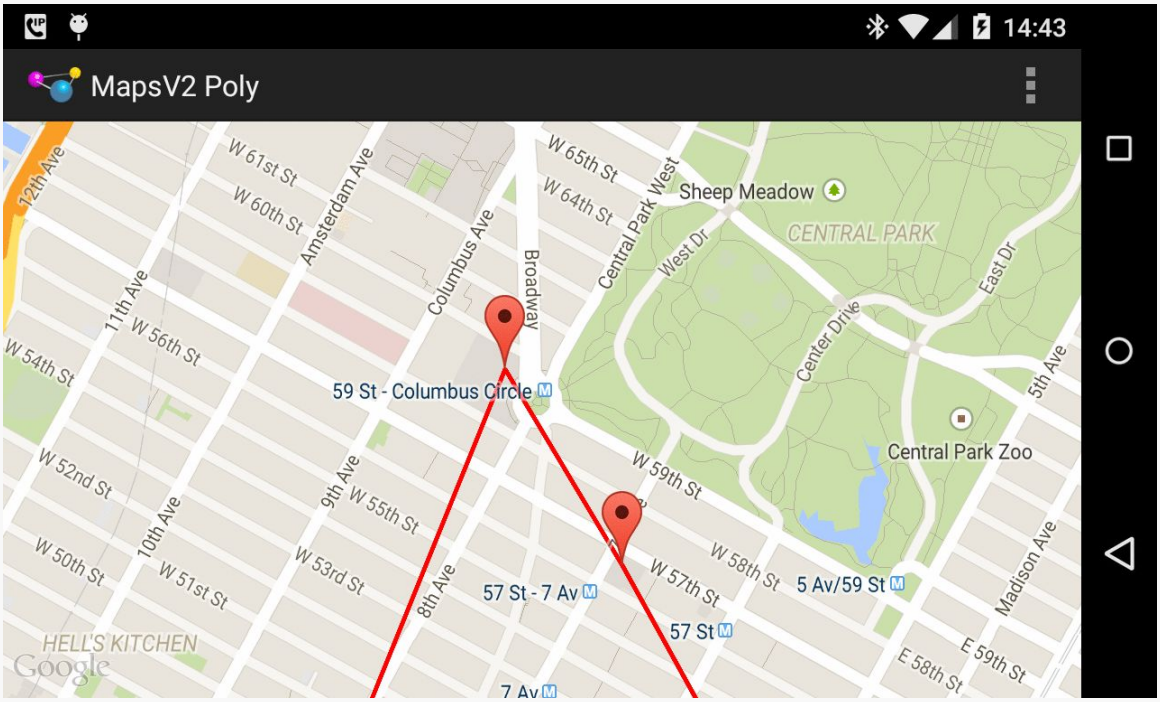


Figure 471: Maps V2 with Polyline

Note that the polyline is drawn using a flat Mercator projection by default. For most maps, that is perfectly fine. If your map will be showing countries and continents, rather than city blocks, you might want to call `geodesic(true)` on the `PolylineOptions`, to have the line drawn on a geodesic curve, reflecting the spherical nature of the Earth (dissenting opinions on that notwithstanding).

Similarly, we create a `PolygonOptions` object, configure it, and pass it to `addPolygon` for our Garment District box. The `add()` method on `PolygonOptions` will take the corners of our polygon, automatically enclosing that region. We also specify the `strokeColor()`. We could have specified a `fillColor()` (default is transparent), `strokeWidth()` (default is 10 pixels), `zIndex()`, and `geodesic()`.

If we run the app and pan the map down to the south a bit, we see our polygon:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

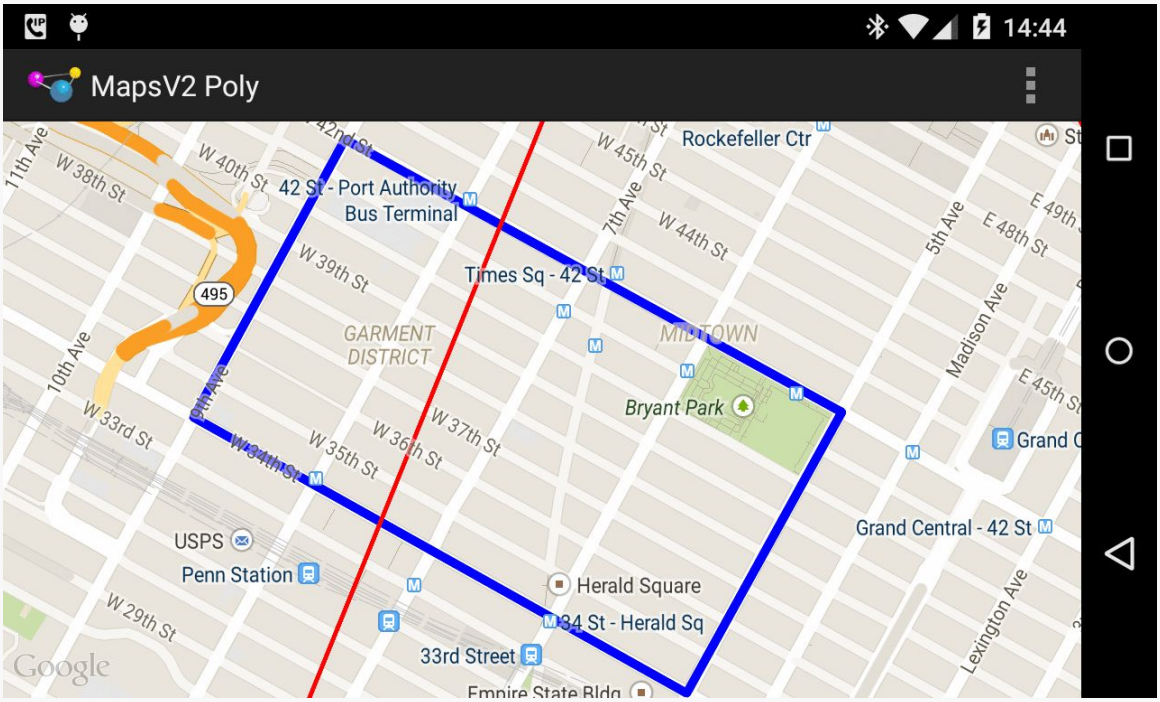


Figure 472: Maps V2 with Polyline and Polygon

As with the polyline, Android automatically handles drawing what is needed based on map center and zoom levels.

Note that, as with markers, we need to re-add the polylines and polygons after a configuration change, as the `GoogleMap` does not retain that information.

Gestures and Controls

By default, standard gestures and controls are enabled on your map:

- The user can change zoom level either by + and - buttons or via “pinch-to-zoom” gestures
- The user can change the center of the map via simple swipe gestures
- The user can change the camera tilt via two-finger vertical swipes, so instead of a traditional top-down perspective, the user can see things on an angle
- The user can change the orientation of the map via a two-finger rotating swipe, to change the typical “north is to the top of the map” to some other orientation

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You can obtain a `UiSettings` object from your `GoogleMap` via `getUiSettings()` to disable these features, if desired:

- `setRotateGesturesEnabled()`
- `setScrollGesturesEnabled()` (for panning the map)
- `setTiltGesturesEnabled()`
- `setZoomControlsEnabled()` (for the + and - buttons)
- `setZoomGesturesEnabled()` (for pinch-to-zoom)

There is also `setAllGesturesEnabled()` to toggle on or off all gesture-based map control. This is roughly analogous to the `android:clickable` attribute on the Maps V1 edition of `MapView`.

There is also `setCompassEnabled()`, to indicate if a compass should be shown if the user changes the map orientation via a rotate gesture.

Tracking Camera Changes

If you have gestures enabled, the user can change the perspective of the map, referred to as changing the camera position. You may need to know about these changes, to perform various operations in your app based upon what is presently visible on the screen. To find out when the camera position changes, you can call `setOnCameraChangeListener()` on the `GoogleMap`, supplying an implementation of `OnCameraChangeListener`, which will be called with `onCameraChange()` as the user pans, zooms, or tilts the map.

To see how this works, we can take a quick peek at the `MapsV2/Camera` sample application, which is a clone of `MapsV2/Popup` with camera position tracking enabled.

Late in `onMapReady()` of `MainActivity`, we call `setOnCameraChangeListener()` on our `GoogleMap`, supplying `MainActivity` itself as the listener:

```
    map.setOnCameraChangeListener(this);
```

This requires `MainActivity` to implement `OnCameraChangeListener` and supply an implementation of `onCameraChange()`:

```
  @Override
  public void onCameraChange(CameraPosition position) {
    Log.d(getClass().getSimpleName(),
          String.format("lat: %f, lon: %f, zoom: %f, tilt: %f",
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                        position.target.latitude,
                        position.target.longitude, position.zoom,
                        position.tilt));
  }
```

Here, we just log a message to LogCat on each camera position change, logging:

- the latitude and longitude of the map center, obtained from the `target` `LatLng` data member of the `CameraPosition` object supplied to `onCameraChange()`,
- the zoom level of the map, from the `zoom` data member of `CameraPosition`, and
- the tilt of the map, in degrees, from the `tilt` data member of `CameraPosition`

As a result, if you run this app and play around with the various gestures, you get a series of LogCat messages with the results:

```
12-26 15:36:39.456: D/MainActivity(31419): lat: 40.763727, lon: -73.983163,
zoom: 15.000000, tilt: 0.000000
12-26 15:36:39.536: D/MainActivity(31419): lat: 40.763797, lon: -73.983118,
zoom: 15.000000, tilt: 0.000000
12-26 15:36:40.796: D/MainActivity(31419): lat: 40.767982, lon: -73.979181,
zoom: 15.000000, tilt: 0.000000
12-26 15:36:41.966: D/MainActivity(31419): lat: 40.766275, lon: -73.981911,
zoom: 15.000000, tilt: 0.000000
12-26 15:36:42.216: D/MainActivity(31419): lat: 40.765145, lon: -73.983651,
zoom: 15.000000, tilt: 0.000000
12-26 15:36:43.526: D/MainActivity(31419): lat: 40.765165, lon: -73.983583,
zoom: 15.000000, tilt: 0.000000
12-26 15:36:44.176: D/MainActivity(31419): lat: 40.765685, lon: -73.981983,
zoom: 15.875862, tilt: 0.000000
12-26 15:36:44.236: D/MainActivity(31419): lat: 40.765685, lon: -73.981983,
zoom: 15.875862, tilt: 0.000000
12-26 15:36:45.566: D/MainActivity(31419): lat: 40.766083, lon: -73.982028,
zoom: 15.875862, tilt: 11.015625
12-26 15:36:45.616: D/MainActivity(31419): lat: 40.766083, lon: -73.982028,
zoom: 15.875862, tilt: 16.171875
12-26 15:36:45.666: D/MainActivity(31419): lat: 40.766083, lon: -73.982028,
zoom: 15.875862, tilt: 24.375000
12-26 15:36:45.726: D/MainActivity(31419): lat: 40.766083, lon: -73.982028,
zoom: 15.875862, tilt: 38.671875
12-26 15:36:45.776: D/MainActivity(31419): lat: 40.766083, lon: -73.982028,
zoom: 15.875862, tilt: 45.234375
12-26 15:36:45.816: D/MainActivity(31419): lat: 40.766083, lon: -73.982028,
zoom: 15.875862, tilt: 48.046875
12-26 15:36:45.846: D/MainActivity(31419): lat: 40.766083, lon: -73.982028,
zoom: 15.875862, tilt: 50.859375
12-26 15:36:45.886: D/MainActivity(31419): lat: 40.766083, lon: -73.982028,
zoom: 15.875862, tilt: 52.968750
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
12-26 15:36:45.926: D/MainActivity(31419): lat: 40.766083, lon: -73.982028,
zoom: 15.875862, tilt: 56.484375
12-26 15:36:45.966: D/MainActivity(31419): lat: 40.766083, lon: -73.982028,
zoom: 15.875862, tilt: 57.890625
12-26 15:36:46.096: D/MainActivity(31419): lat: 40.766083, lon: -73.982028,
zoom: 15.875862, tilt: 59.296875
```

Maps in Fragments and Pagers

One key limitation of Maps V1 was that you could only have one `MapView` instance per *process*. Presumably, the proprietary code at the heart of the Maps SDK add-on used static data members for some state management, ones that would get messed up if there were two or more `MapView` widgets in active use.

Fortunately, Maps V2 gets rid of this restriction. You are welcome to have multiple `MapFragment` objects if that makes sense. Maps are relatively memory-intensive, so you should not be planning on having dozens or hundreds of them in use at a time, but you can have more than one.

To showcase this, the `MapsV2/Pager` sample application hosts 10 `MapFragment` instances as pages in a `ViewPager`. The bulk of the application is a clone of one of the `ViewPager` samples from the chapter on `ViewPager`.

Having maps in a `ViewPager` presents a bit of a problem, in terms of interpreting horizontal swipe events. Normally, `ViewPager` handles those itself. However, that would mean that the user cannot pan the map horizontally, which makes using the map somewhat challenging. In this sample, we will augment the `ViewPager` with logic to allow horizontal swiping on the maps and on the tab strip.

Our activity inflates a layout that contains our `ViewPager` along with a `PagerTabStrip`:

```
<?xml version="1.0" encoding="utf-8"?>
<com.commonsware.android.mapsv2.pager.MapAwarePager
xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/pager"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <android.support.v4.view.PagerTabStrip
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="top"/>

</com.commonsware.android.mapsv2.pager.MapAwarePager>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, you will note that this is not ViewPager, but rather MapAwarePager, a custom subclass of ViewPager that we will examine shortly.

MainActivity then populates the MapAwarePager with an instance of a MapPageAdapter:

```
package com.commonsware.android.mapsv2.pager;

import android.os.Bundle;
import android.support.v4.view.PagerAdapter;
import android.support.v4.view.ViewPager;

public class MainActivity extends AbstractMapActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (readyToGo()) {
      setContentView(R.layout.activity_main);

      ViewPager pager=(ViewPager)findViewById(R.id.pager);

      pager.setAdapter(buildAdapter());
    }
  }

  private PagerAdapter buildAdapter() {
    return(new MapPageAdapter(this, getFragmentManager()));
  }
}
```

MapPageAdapter is a FragmentStatePagerAdapter, not a FragmentPagerAdapter. This means that as the user swipes through our ViewPager, the adapter has the right to discard old fragments when it creates new ones. This helps reduce the overall memory footprint of our activity.

```
package com.commonsware.android.mapsv2.pager;

import android.content.Context;
import android.app.Fragment;
import android.app.FragmentManager;
import android.support.v13.app.FragmentStatePagerAdapter;

public class MapPageAdapter extends FragmentStatePagerAdapter {
  Context ctxt=null;

  public MapPageAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);
    this.ctxt=ctxt;
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public int getCount() {
    return(10);
  }

  @Override
  public Fragment getItem(int position) {
    return(new PageMapFragment());
  }

  @Override
  public String getPageTitle(int position) {
    return(ctxt.getString(R.string.map_page_title) + String.valueOf(position +
1));
  }
}
```

`MapPageAdapter` declares that there should be ten pages (in `getCount()`) and returns an instance of `PageMapFragment` for each page. `PageMapFragment` is a subclass of `MapFragment`, and so is responsible for displaying our map:

```
package com.commonsware.android.mapsv2.pager;

import android.os.Bundle;
import com.google.android.gms.maps.CameraUpdate;
import com.google.android.gms.maps.CameraUpdateFactory;
import com.google.android.gms.maps.GoogleMap;
import com.google.android.gms.maps.MapFragment;
import com.google.android.gms.maps.OnMapReadyCallback;
import com.google.android.gms.maps.model.LatLng;
import com.google.android.gms.maps.model.MarkerOptions;

public class PageMapFragment extends MapFragment implements
    OnMapReadyCallback {
  private boolean needsInit=false;

  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);

    if (savedInstanceState == null) {
      needsInit=true;
    }

    getMapAsync(this);
  }

  @Override
  public void onMapReady(final GoogleMap map) {
    if (needsInit) {
      CameraUpdate center=
          CameraUpdateFactory.newLatLng(new LatLng(40.76793169992044,
                                                   -73.98180484771729));
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      CameraUpdate zoom=CameraUpdateFactory.zoomTo(15);

      map.moveCamera(center);
      map.animateCamera(zoom);
    }

    addMarker(map, 40.748963847316034, -73.96807193756104, R.string.un,
              R.string.united_nations);
    addMarker(map, 40.76866299974387, -73.98268461227417,
              R.string.lincoln_center, R.string.lincoln_center_snippet);
    addMarker(map, 40.765136435316755, -73.97989511489868,
              R.string.carnegie_hall, R.string.practice_x3);
    addMarker(map, 40.70686417491799, -74.01572942733765,
              R.string.downtown_club, R.string.heisman_trophy);
  }

  private void addMarker(GoogleMap map, double lat, double lon,
                         int title, int snippet) {
    map.addMarker(new MarkerOptions().position(new LatLng(lat, lon))
                                     .title(getString(title))
                                     .snippet(getString(snippet)));
  }
}
```

If we simply wanted to display an unconfigured map, we could just have `MapPageAdapter` create and return instances of `MapFragment` directly. If we want to configure our map, though, we need to get control when the `GoogleMap` object is ready for use. One way to do that is to extend `MapFragment` and override `onActivityCreated()` and call `getMapAsync()` there to begin the whole get-the-`GoogleMap`-loaded process. In `onMapReady()`, we can then go ahead and configure the map much as we have done in previous examples, just from within the fragment itself rather than from the hosting activity.

`MapAwarePager` overrides one key method of `ViewPager`: `canScroll()`:

```
package com.commonsware.android.mapsv2.pager;

import android.content.Context;
import android.support.v4.view.PagerTabStrip;
import android.support.v4.view.ViewPager;
import android.util.AttributeSet;
import android.view.SurfaceView;
import android.view.View;

public class MapAwarePager extends ViewPager {
  public MapAwarePager(Context context, AttributeSet attrs) {
    super(context, attrs);
  }

  @Override
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  protected boolean canScroll(View v, boolean checkV, int dx, int x,
                              int y) {
    if (v instanceof SurfaceView || v instanceof PagerTabStrip) {
      return(true);
    }

    return(super.canScroll(v, checkV, dx, x, y));
  }
}
```

`canScroll()` should return `true` if the `View` (and specifically the supplied X and Y coordinates within that `View`) can be scrolled horizontally, `false` otherwise. In our case, we want to say that the map and the tab strip are each scrollable horizontally. As it turns out, the passed-in `View` for our `MapFragment` will be the map if it is a subclass of `SurfaceView` (determined by trial and error on the author's part, with hopes for a more authoritative solution in a future edition of the Maps V2 API). So, if the passed-in `View` is either a `SurfaceView` or a `PagerTabStrip`, we return `true`, otherwise we default to normal logic.

The result is a series of independent maps, one per page:

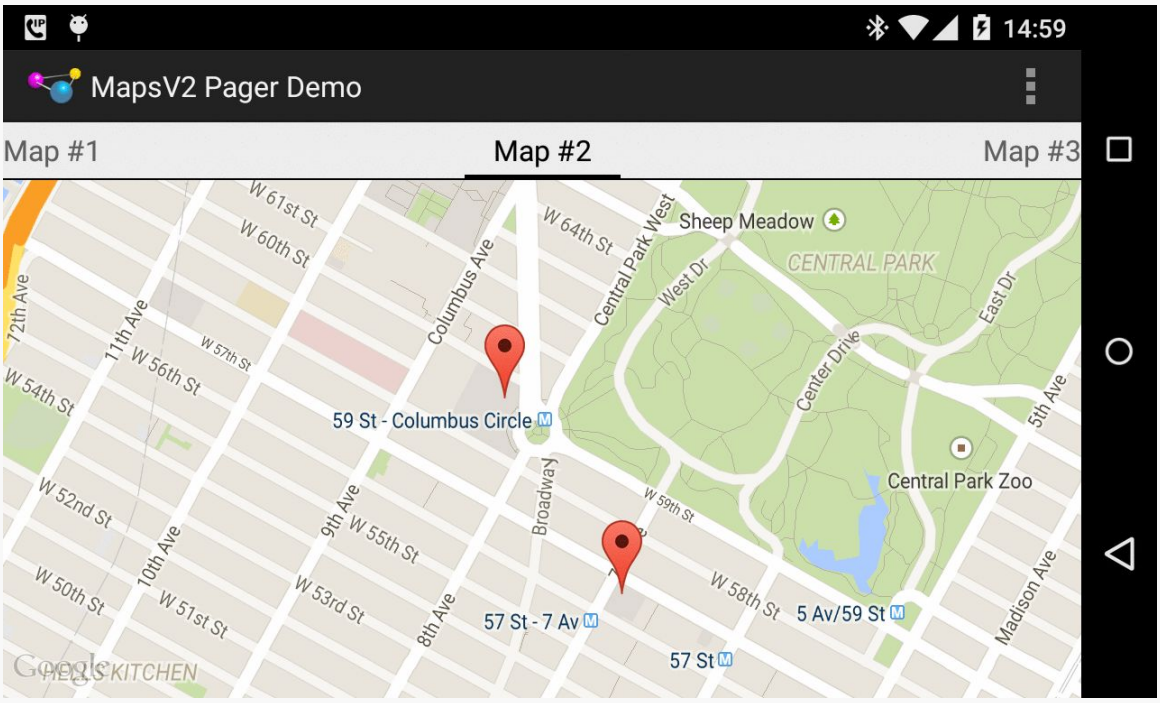


Figure 473: Multiple Maps V2 Maps in a ViewPager

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Each map is independent: if the user pans or zooms one map, that has no impact on any of the other pages. Panning the maps horizontally works; to move between pages, use the tab strip.

Animating Marker Movement

Markers, by default, are static, unless you make them be draggable, and then only the user can drag them.

However, you are welcome to update the position of a `Marker` at any point, by calling `setPosition()` and supplying a new `LatLng`. The `Marker` then will jump to that position.

But what if you want to animate the movement of a `Marker` from its current position to a new one? Maps V2 does not offer anything "out of the box" for implementing this, but Google demonstrated approaches for this in a "DevBytes" video and related bit of code in a GitHub Gist. This section will cover the technique appropriate for API Level 14+, including a full working sample (the Gist shows code but not its usage).

Problem #1: Animating a LatLng

The position of a `Marker` is a `LatLng`, as we have seen previously. `LatLng` is not a simple number, and so the animator framework needs our assistance to animate them. Specifically, we need a `TypeEvaluator` for `LatLng`, with our `evaluate()` method taking the initial and end positions and computing another `LatLng` representing the `fraction` position between those other positions. This concept was introduced back in the chapter on the animator framework.

A simple approach to computing the fractional `LatLng` would be to apply the fraction to the latitude and the longitude as Java double values:

```
LatLng interpolate(float fraction, LatLng initial, LatLng end) {
  double lat = (end.latitude - initial.latitude) * fraction + initial.latitude;
  double lng = (end.longitude - initial.longitude) * fraction +
initial.longitude;

  return(new LatLng(lat, lng));
}
```

That would work reasonably well for fairly close points, such as animating a marker within a city. However, animating markers across longer distances means that we

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

have to take into account some geographic realities that a simple calculation will miss.

Problem #2: The Earth Is Not Flat (Really!)

One bit of reality is that the Earth is round. The above calculation assumes that the Earth is flat. Calculating "great circle" positions requires a fair bit of spherical trigonometry, known to cause loss of hair in software developers.

Hence, ideally, we will use somebody's existing debugged algorithm for that.

Problem #3: 180 Equals –180, At Least For Longitude

The other problem is that longitudes wrap around, as 180 degrees longitude is equivalent to –180 degrees longitude, and longitudinal values are considered to be between 180 and –180. In cases where we would not cross 180 degrees longitude, this is not an issue. However, a simple calculation might miss this and wind up having our animation "take the long way" (e.g., animating from –175 degrees longitude to 175 degrees longitude by going 350 degrees around the Earth, rather than just 10 degrees and crossing the International Date Line).

Introducing Some Googly Assistance

Google themselves have released a utility library for Maps V2. It offers polyline and polygon decoding, primarily for interoperability with other location-related Google services like the Google Directions API. The `SphericalUtil` class handles all of the nasty math for computing distances along the surface of the Earth and related calculations. It also offers `BubbleIconFactory`, which makes it easy to create marker icons that look a bit like info windows (complete with border and caret) wrapping around a bit of text or an icon.

In our case, we can use `SphericalUtil` to handle Problem #2 and Problem #3, interpolating the location between two `LatLng` values, taking the curvature of the Earth and longitude idiosyncrasies into account.

Seeing This in Action

The `MapsV2/Animator` sample project is a modified version of the `MapsV2/Markers` project, adding in the notion of animating a marker from its original position (Lincoln Center) to a new position (Penn Station) within Manhattan.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Since we want to use the Google map utility library, we need to add it as a dependency.

- Android Studio users can simply add `compile 'com.google.maps.android:android-maps-utils:0.3.4'` (or a higher version) to the `dependencies` closure.
- Eclipse users will need to download and add it to the project by one means or another. Note, though, that this library project depends upon the Play Services SDK. Hence, you will need to update the project's configuration to have it reference your development environment's copy of the Play Services SDK. Then, when you attach the library project to your project, you can remove your own reference to the Play Services SDK, as you will get that through the utility library.

We need to know where our starting and ending position for the animation will be, in terms of `LatLng` objects. Since those have no dependencies upon a `Context` or anything, we can simply declare them as `static final` values:

```
private static final LatLng PENN_STATION=new LatLng(40.749972,
                                                    -73.992319);
private static final LatLng LINCOLN_CENTER=
    new LatLng(40.76866299974387, -73.98268461227417);
```

We will also need the actual `Marker` object created when we add our starting position (`LINCOLN_CENTER`) to the map. So far, we have ignored the `Marker` returned by `addMarker()` on `GoogleMap`, but now we need that. So, our own `addMarker()` method now returns this value:

```
private Marker addMarker(GoogleMap map, double lat, double lon,
                         int title, int snippet) {
  return(map.addMarker(new MarkerOptions().position(new LatLng(lat,
                                                                lon))
                                          .title(getString(title))
                                          .snippet(getString(snippet))));
}
```

We also now have a `markerToAnimate` data member of the activity, for our `Marker`, which we populate from our modified `addMarker()` method:

```
  addMarker(map, 40.748963847316034, -73.96807193756104,
            R.string.un, R.string.united_nations);
  markerToAnimate=
      addMarker(map, LINCOLN_CENTER.latitude,
                LINCOLN_CENTER.longitude, R.string.lincoln_center,
                R.string.lincoln_center_snippet);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    addMarker(map, 40.765136435316755, -73.97989511489868,
              R.string.carnegie_hall, R.string.practice_x3);
    addMarker(map, 40.70686417491799, -74.01572942733765,
              R.string.downtown_club, R.string.heisman_trophy);
```

To make the sample work repeatedly, it would be nice to support bi-directional animation, starting with animating from Lincoln Center to Penn Station, then reversing the animation to go back to Lincoln Center. That means that we need to know, for any particular animation, where the end position should be. So, we track a `LatLng` for the next end position, surprisingly named `nextAnimationEnd`, initializing it to be `PENN_STATION` (since we are starting at the outset at `LINCOLN_CENTER`):

```
  private LatLng nextAnimationEnd=PENN_STATION;
```

Next, we need to give the user a means of actually requesting the animation to run. To do that, we define a new menu XML resource for an `animate` menu item (using the `directions` icon for lack of a better handy icon):

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/animate"
    android:icon="@android:drawable/ic_menu_directions"
    android:showAsAction="ifRoom"
    android:title="@string/animate"/>

</menu>
```

We then load that menu resource in an overridden `onCreateOptionsMenu()` and direct the click event to an `animateMarker()` method in `onOptionsItemSelected()`:

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.animate, menu);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.animate) {
      animateMarker();

      return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In `animateMarker()`, we need to do two things:

1. Actually run the animation
2. Ensure that the camera position is such that the animation will actually be visible, as it is pointless to animate a marker between two points if the currently-viewed portion of the map does not show those points

To handle the camera position, we need to use `moveCamera()` with a `CameraUpdate` from `CameraUpdateFactory`, as we used to set the initial camera position and zoom level. To handle the case where we want one or more points to be visible, we can use the `newLatLngBounds()` method on `CameraUpdateFactory`. This takes a `LatLngBounds` describing the area that needs to be visible, plus a padding amount in pixels for where that area should be inset within the map.

Of course, this implies that we have a `LatLngBounds`.

Since `LatLngBounds` also does not depend upon a `Context` or much of anything, we can define one of those as a `static final` data member, using a `LatLngBounds.Builder` instance:

```
  private static final LatLngBounds bounds=
      new LatLngBounds.Builder().include(LINCOLN_CENTER)
                                .include(PENN_STATION).build();
```

A `LatLngBounds.Builder` takes one or more `LatLng` objects — passed in via `include()` – then constructs a `LatLngBounds` that encompasses all of those points via `build()`.

Our `animateMarker()` method then starts off by using `moveCamera()` to reset the camera to show that defined region:

```
  private void animateMarker() {
    map.moveCamera(CameraUpdateFactory.newLatLngBounds(bounds, 48));

    Property<Marker, LatLng> property=
        Property.of(Marker.class, LatLng.class, "position");
    ObjectAnimator animator=
        ObjectAnimator.ofObject(markerToAnimate, property,
                                new LatLngEvaluator(), nextAnimationEnd);
    animator.setDuration(2000);
    animator.start();

    if (nextAnimationEnd == LINCOLN_CENTER) {
      nextAnimationEnd=PENN_STATION;
    }
    else {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      nextAnimationEnd=LINCOLN_CENTER;
    }
  }
```

Then, we need to set up the animation. To do this, we will use the object animator framework, specifically an `ObjectAnimator`. We know the `Marker` that we want to animate (`markerToAnimate`) and we know where we want to animate it to (`nextAnimationEnd`). What we need is to indicate the property to animate on this object, plus provide help to actually animate a `LatLng`.

To specify the property, we could just pass in the name of the property (`"position"`). However, in `animateMarker()`, we set up a `Property` object via the static `of()` factory method. This makes our use of `ofObject()` more type-safe, as `Property` will help enforce that we are animating a `Marker` using `LatLng` values.

To animate `LatLng` values, we need a `TypeEvaluator` for `LatLng`, here defined as a static inner class named `LatLngEvaluator`:

```
  private static class LatLngEvaluator implements TypeEvaluator<LatLng> {
    @Override
    public LatLng evaluate(float fraction, LatLng startValue,
                           LatLng endValue) {
      return(SphericalUtil.interpolate(startValue, endValue, fraction));
    }
  }
```

Our `evaluate()` method turns around and calls the static `interpolate()` method on `SphericalUtil`, supplied by Google's map utility library. `interpolate()` handles all the nasty spherical trigonometry and stuff, so we do not have to.

We then set the duration of the animation to be two seconds, and start the animation.

Finally, to reverse the animation for the next request, `animateMarker()` resets the value of `nextAnimationEnd` to be `PENN_STATION` or `LINCOLN_CENTER`, wherever we will animate to next.

This version of the app starts off as do all the others, except for the new action bar item:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

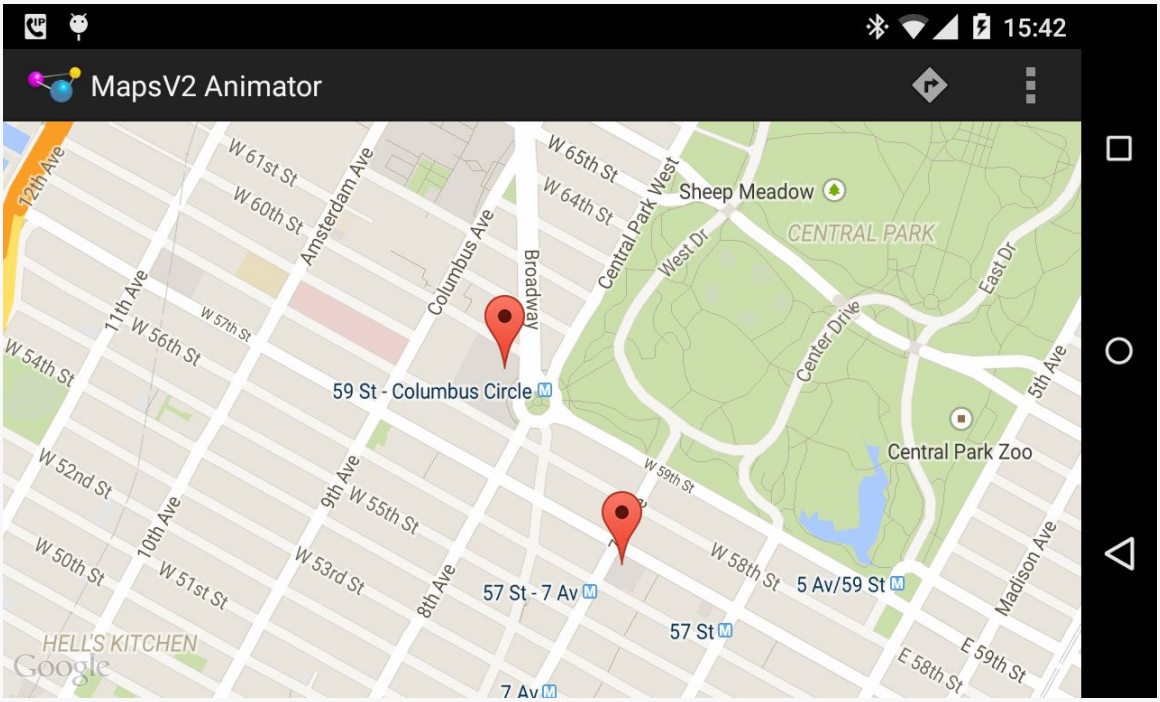


Figure 474: Maps V2 Animator Demo, As Initially Launched

Tapping that action bar item (“directions” icon) will reset the camera position and start animating the marker:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

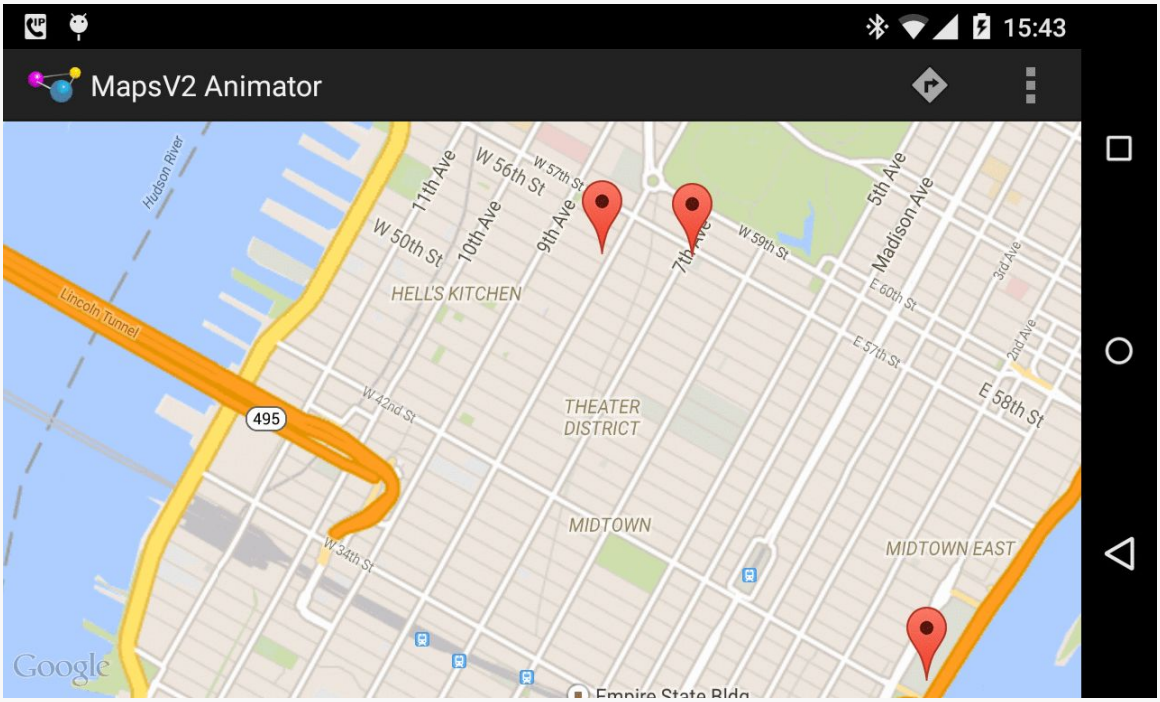


Figure 475: Maps V2 Animator Demo, Partially Through an Animation

Two seconds later, the marker will reach its destination, presumably to board a train:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

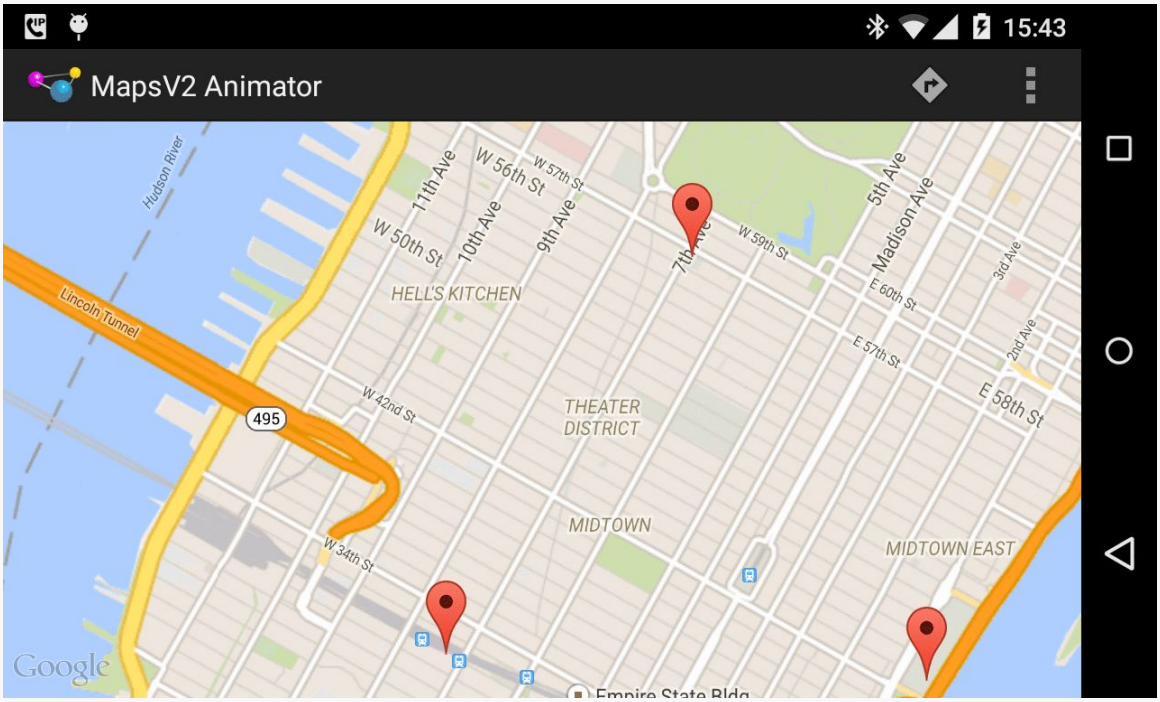


Figure 476: Maps V2 Animator Demo, with Marker Animated to Penn Station

Honoring Traffic Rules, Like "Drive Only On Streets"

You will notice that our animation ignores other aspects of reality, such as buildings that might be in the way. Sometimes, that is appropriate, such as animating the movement of:

- a bird
- a plane
- a costumed superhero with independent flight capability

Sometimes, though, we need to take into account those obstacles, such as animating the movement of:

- a pedestrian
- a car
- a costumed superhero "flying" by means of swinging between buildings using dynamically-generated cables of either natural or synthetic origin

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, to do this implies that we know where the obstacles are. Or, more accurately, we would need to animate the marker along known good waypoints, such as streets.

The animation would not be especially difficult, as `ofObject()` can take a series of waypoints. However, we would need to find those waypoints, and there is nothing in Maps V2 itself that supplies this data.

Maps, of the Indoor Variety

The good news is that Maps V2 supports Google's indoor maps, for those venues for which Google has indoor map data.

The bad news is that for some reason, only one map at a time supports indoor maps. The default will be that the first map you create will support indoor maps, and others will not.

To see if a given map offers indoor map capability, you can call `isIndoorEnabled()` on `GoogleMap`. To toggle this capability, call `setIndoorEnabled()`.

Taking a Snapshot of a Map

Once a map is drawn, you can take a snapshot of it, converting the viewed map into a `Bitmap` object. This is designed to take an image of the map and use it in places where a `MapFragment`, or even a `MapView`, cannot go, such as:

- Things tied to a `RemoteViews`, such as a custom `Notification`
- Thumbnails of maps, for an app that allows users to manipulate several maps at once

The `GoogleMap` object has two flavors of a `snapshot()` method. Both take a `SnapshotReadyCallback` object. You will need to supply an instance of something implementing the `SnapshotReadyCallback` interface, overriding `onSnapshotReady()`, where you will receive your `Bitmap`.

One flavor of `snapshot()` takes just the `SnapshotReadyCallback`; the other also takes a `Bitmap` of the proper dimensions, such as a previous snapshot `Bitmap` that you want to recycle. Using the latter `snapshot()` is recommended where possible, so you do not need to allocate new `Bitmap` objects on each `snapshot()` call.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that `snapshot()` will only work once the map is actually rendered. So, for example, calling `snapshot()` from `onCreate()` of your activity will fail, because the map has not been rendered yet. `snapshot()` is designed to be called based upon user input, either to manually capture a snapshot or based on navigation (e.g., tapping on a `ListView` item triggers saving a snapshot of the current map as a thumbnail before changing the map contents).

Also, the documentation for `snapshot()` contains the following:

> **Note**: Images of the map must not be transmitted to your servers, or otherwise used outside of the application. If you need to send a map to another application or user, send data that allows them to reconstruct the map for the new user instead of a snapshot.

As this statement may be tied to the terms and conditions of your use of Maps V2, you should talk with qualified legal counsel before:

- Saving a snapshot to external storage
- Sharing a snapshot via `ACTION_SEND`
- Sending a snapshot to your server

or similar operations.

MapFragment vs. MapView

So far, all the examples shown in this chapter use `MapFragment`. In most cases, this is the right thing to use.

However, there may be places where you really want to use a `View`, rather than a `Fragment`, for your maps.

The good news is that Maps V2 does have a `MapView`. `MapFragment` usually handles creating and managing the `MapView` for you, but you can, if you wish, eschew `MapFragment` and manage the `MapView` yourself.

The biggest limitation is that you need to forward the lifecycle methods from your activity or fragment on to the `MapView`, calling `onCreate()`, `onResume()`, `onPause()`, `onDestroy()`, and `onSaveInstanceState()` on the `MapView`. Normally, `MapFragment` would do that for you, saving you the trouble.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Also note that while `MapView` is a `ViewGroup`, you are not allowed to add child widgets to it.

About That AbstractMapActivity Class…

Early on, we hand-waved our way past the `AbstractMapActivity` that all of our `MainActivity` classes inherit from, and we skirted past the `readyToGo()` method that we were calling. Also, you may have noticed that our app has an action bar overflow item, that we do not seem to be creating in `MainActivity`.

Now, it is time to dive into what is going on in our `AbstractMapActivity` implementations.

Getting Maps V2 Ready to Go

The `readyToGo()` method in `AbstractMapActivity` is designed to help us determine if Maps V2 is "ready to go" and, if not, to help the user perhaps fix their device such that Maps V2 will work in the future:

```
protected boolean readyToGo() {
  int status=
      GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);

  if (status == ConnectionResult.SUCCESS) {
    if (getVersionFromPackageManager(this) >= 2) {
      return(true);
    }
    else {
      Toast.makeText(this, R.string.no_maps, Toast.LENGTH_LONG).show();
      finish();
    }
  }
  else if (GooglePlayServicesUtil.isUserRecoverableError(status)) {
    ErrorDialogFragment.newInstance(status)
                       .show(getFragmentManager(),
                             TAG_ERROR_DIALOG_FRAGMENT);
  }
  else {
    Toast.makeText(this, R.string.no_maps, Toast.LENGTH_LONG).show();
    finish();
  }

  return(false);
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

First, we call the static `isGooglePlayServicesAvailable()` method on `GooglePlayServicesUtil`. This will return an integer indicating whether Maps V2 is available for our use or not.

If the return value is `ConnectionResult.SUCCESS` — meaning Maps V2 is indeed available to us – we check to see if OpenGL ES is version 2.0 or higher, as we did not require that in the manifest. There are a few ways in Android to check the OpenGL ES version. This sample uses some code from the Compatibility Test Suite (CTS), examining `PackageManager` to determine the major level:

```
  // following from
  // https://android.googlesource.com/platform/cts/+/master/tests/tests/graphics/
src/android/opengl/cts/OpenGlEsVersionTest.java

  /*
   * Copyright (C) 2010 The Android Open Source Project
   *
   * Licensed under the Apache License, Version 2.0 (the
   * "License"); you may not use this file except in
   * compliance with the License. You may obtain a copy of
   * the License at
   *
   * http://www.apache.org/licenses/LICENSE-2.0
   *
   * Unless required by applicable law or agreed to in
   * writing, software distributed under the License is
   * distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
   * CONDITIONS OF ANY KIND, either express or implied. See
   * the License for the specific language governing
   * permissions and limitations under the License.
   */

  private static int getVersionFromPackageManager(Context context) {
    PackageManager packageManager=context.getPackageManager();
    FeatureInfo[] featureInfos=
        packageManager.getSystemAvailableFeatures();
    if (featureInfos != null && featureInfos.length > 0) {
      for (FeatureInfo featureInfo : featureInfos) {
        // Null feature name means this feature is the open
        // gl es version feature.
        if (featureInfo.name == null) {
          if (featureInfo.reqGlEsVersion != FeatureInfo.GL_ES_VERSION_UNDEFINED)
{
            return getMajorVersion(featureInfo.reqGlEsVersion);
          }
          else {
            return 1; // Lack of property means OpenGL ES
                      // version 1
          }
        }
      }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
    return 1;
  }

  /** @see FeatureInfo#getGlEsVersion() */
  private static int getMajorVersion(int glEsVersion) {
    return((glEsVersion & 0xffff0000) >> 16);
  }
```

If the major version is `2` (or, theoretically, higher), we return `true` from `readyToGo()`, so `MainActivity` knows to continue on setting up the map. If the major version is `1`, we display a `Toast` — a production-grade app would do something else to let the user know of the problem, most likely.

But, what if `isGooglePlayServicesAvailable()` returns something else?

There are two major possibilities here:

1. The error is something that the user might be able to rectify, such as by downloading the Google Play Services app from the Play Store
2. The error is something that the user cannot recover from

We can distinguish these two cases by calling the static `isUserRecoverableError()` on `GooglePlayServicesUtil`, passing in the value we received from `isGooglePlayServicesAvailable()`. This will return `true` if the user might be able to fix the problem, `false` otherwise.

In the `false` case, the user is just out of luck, so we display a `Toast` to alert them of this fact, then `finish()` the activity and return `false`, so `MainActivity` skips over the rest of its work.

In the `true` case, we can display something to the user to prompt them to fix the problem. One way to do that is to use a dialog obtained from Google code, by calling the static `getErrorDialog()` method on the `GooglePlayServicesUtil` class. In our case, we wrap that in a `DialogFragment` named `ErrorDialogFragment`, implemented as a static inner class of `AbstractMapActivity`:

```
  public static class ErrorDialogFragment extends DialogFragment {
    static final String ARG_STATUS="status";

    static ErrorDialogFragment newInstance(int status) {
      Bundle args=new Bundle();

      args.putInt(ARG_STATUS, status);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    ErrorDialogFragment result=new ErrorDialogFragment();

    result.setArguments(args);

    return(result);
  }

  @Override
  public Dialog onCreateDialog(Bundle savedInstanceState) {
    Bundle args=getArguments();

    return GooglePlayServicesUtil.getErrorDialog(args.getInt(ARG_STATUS),
                                                 getActivity(), 0);
  }

  @Override
  public void onDismiss(DialogInterface dlg) {
    if (getActivity() != null) {
      getActivity().finish();
    }
  }
}
```

While the code and comments around `getErrorDialog()` suggest that there is some way for us to find out if the user performed actions that fix the problem, this code does not seem to work well in practice. After all, downloading Google Play Services is asynchronous, so even if the user returns to our app, it is entirely likely that Maps V2 is still unavailable. As a result, when the user is done with the dialog, we `finish()` the activity, forcing the user to start it again if and when they are done downloading Google Play Services.

Testing this code requires an older device, one in which the "Google Play services" app can be uninstalled... if it can be installed at all.

As it turns out, not all Android devices support the Play Store, or the Google Play Services by extension. Notably, if the device lacks the Play Store, `isUserRecoverableError()` returns `true`, even though the user cannot recover from this situation (except perhaps via a firmware update).

(An earlier problem where `getErrorDialog()` could return `null` even for cases where the error is supposedly user-recoverable has been fixed)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Handling the License Terms

`AbstractMapActivity` has implementations of `onCreateOptionsMenu()` and `onOptionsItemSelected()` that will add a "Legal Notices" item to the overflow menu and bring up `LegalNoticesActivity` when that menu item is tapped:

```
package com.commonsware.android.mapsv2.basic;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import com.google.android.gms.common.GooglePlayServicesUtil;

public class LegalNoticesActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.legal);

    TextView legal=(TextView)findViewById(R.id.legal);

    legal.setText(GooglePlayServicesUtil.getOpenSourceSoftwareLicenseInfo(this));
  }
}
```

`LegalNoticesActivity` simply has a `TextView` inside of a `ScrollView` and fills in the `TextView` with the results of calling `getOpenSourceSoftwareLicenseInfo()` on `GooglePlayServicesUtil`. This method returns the legalese that you need to display to the users from somewhere in your app.

Helper Libraries for Maps V2

Many developers have been busy writing libraries that help in the development of Maps V2 applications, beyond Google's own utility library mentioned in the section on animating markers.

Perhaps the most expansive of these is the Android Maps Extensions library, by Maciej Górski. The big thing that this library offers is marker clustering, where as the user zooms out, individual markers are replaced by a marker representing a cluster, so you avoid flooding a small area with too many individual markers:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 477: Map with Many Markers (from Android Maps Extensions demo app)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

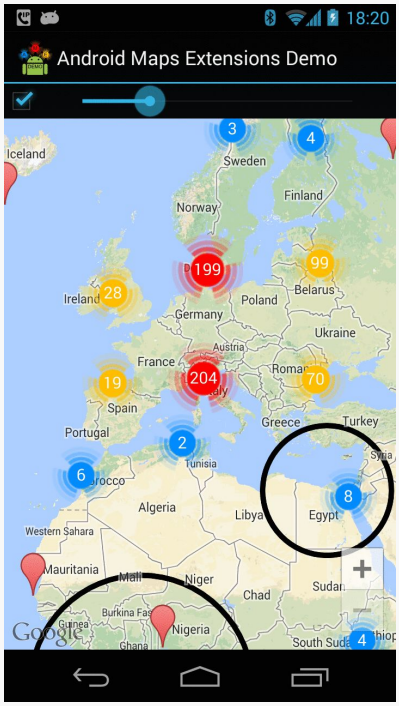


Figure 478: Same Map with Cluster Markers (from Android Maps Extensions demo app)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

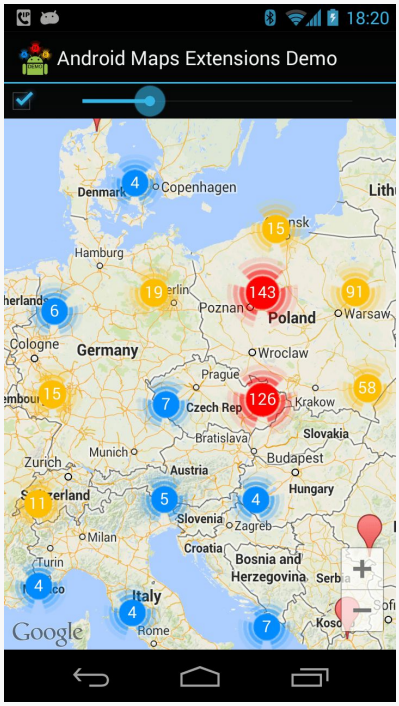


Figure 479: Same Map with Zoomed In Cluster Markers (from Android Maps Extensions demo app)

This library wraps the Maps V2 classes, allowing the library to offer extensions to the standard Maps V2 API, including:

- Associating your own data with `Marker`, `Polygon`, `Polyline`, and other classes, to tie them back to your models
- Getters to retrieve previously-defined markers, etc.
- Etc.

Another library offering marker clustering is clusterkraf, from Two Toasters.

The clusterkraf library can optionally integrate with Cyril Mottier's Polaris2 library. His original Polaris library aimed to provide more features to Maps V1; Polaris2 fills a similar role for Maps V2. At this time, Polaris2 is a smaller library, simply because Maps V2 handles much of what Polaris provided. Polaris2, like Android Maps Extensions, wraps the Maps V2 API with its own classes, in lieu of subclassing (since most Maps V2 classes are marked `final`). Of note, Polaris2 offers `reset()` methods on many of the `...Options` classes (e.g., `MarkerOptions`), and offers constants for the minimum and maximum valid latitude and longitude.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Problems with Maps V2 at Runtime

Portions of the logic that powers your Maps V2 `MapFragment` are supplied by the Google Play Services app. As a result, many operations with Maps V2, such as manipulating markers, require IPC calls between your app and Google Play Services. If those IPC calls are synchronous, they will add a bit of overhead to your app — enough that you will want to avoid them in time-critical pieces of code, tight loops, and the like.

Problems with Maps V2 Deployment

Of course, the key question is: should you be using Maps V2 at all?

Google thinks so, as they have turned off access to new API keys for Maps V1. That makes ongoing development of Maps V1 solutions a bit risky, as you cannot create new API keys for new signing keys, such as if you need to replace your debug keystore.

However, Maps V2 has some deployment limitations at this time. While 99.8+% of Android devices that have the Play Store have the requisite OpenGL ES 2.0+, some devices that have a suitable OpenGL ES version may not have the Play Store or otherwise be unable to get Google Play Services, required for using Maps V2. The `isGooglePlayServicesAvailable()` approach advocated by Google can help determine this at runtime, though this approach used to have some bugs, and it still cannot always help you recover from this problem.

What Non-Compliant Devices Show

If your app tries to bring up Maps V2 on a device that cannot possibly have the Play Services Framework — such as a Kindle Fire — the user will see an error dialog:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

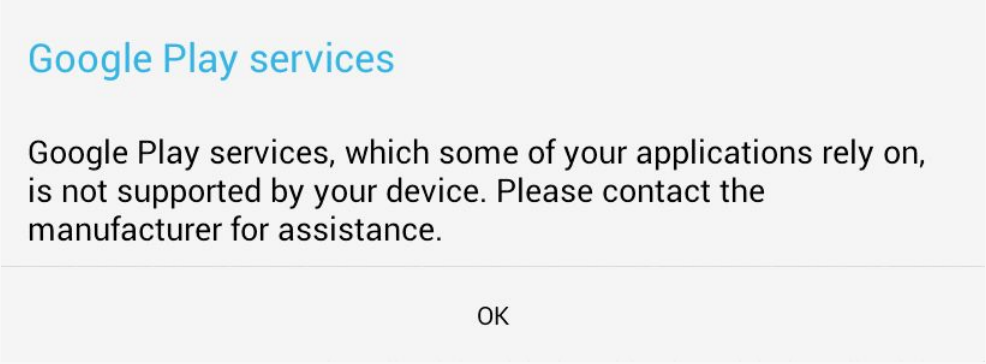


Figure 480: Maps V2 Error on Kindle Fire

Mapping Alternatives

Beyond using Maps V2 or Maps V1, you may need to consider other mapping alternatives. The Google mapping APIs are only available on Android devices that have the Maps SDK add-on (Maps V1) or Google Play Services (Maps V2). Not all devices have those. And, the limitations of Maps V2 deployment and the deprecation of Maps V1 may convince you that relying upon Google for maps is not safe at the present time.

The most common native replacement for Google's mapping is OpenStreetMap, which to some extent is "the Wikipedia of maps". OSMDroid is a library that provides a Maps V1-ish API for embedding OpenStreetMap-based maps into your application.

Another solution is to integrate Web-based Google maps into your app, the same way that you might embed them into your Web site. An activity hosting a `WebView` can display a Web-based Google Map, for example.

Certain devices may have access to other native mapping solutions. For example, Amazon has published their own maps API for use with the Kindle Fire.

News and Getting Help

The Maps V2 team maintains a set of release notes for when they ship updates to the Maps V2 support in the Play Services library project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The official support point for Maps V2 for Android is Stack Overflow. Questions tagged with both `android` and `google-maps` should show up on Google's radar.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Crafting Your Own Views

One of the classic forms of code reuse is the GUI widget. Since the advent of Microsoft Windows — and, to some extent, even earlier – developers have been creating their own widgets to extend an existing widget set. These range from 16-bit Windows "custom controls" to 32-bit Windows OCX components to the innumerable widgets available for Java Swing and SWT, and beyond. Android lets you craft your own widgets as well, such as extending an existing widget with a new UI or new behaviors.

Note that the material in this chapter is focused on creating custom `View` classes for use within a single Android project. If your goal is to truly create reusable custom widgets, you will also need to learn how to package them so they can be reused — that is covered in a later chapter.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Pick Your Poison

You have five major options for creating a custom `View` class.

First, your "custom `View` class" might really only be custom `Drawable` resources. Many widgets can adopt a radically different look and feel just with replacement graphics. For example, you might think that these toggle buttons from the Android 2.1 Google Maps application are some fancy custom widget:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 481: Google Maps navigation toggle buttons

In reality, those are just radio buttons with replacement images.

Second, your custom `View` class might be a simple subclass of an existing widget, where you override some behaviors or otherwise inject your own logic. Unfortunately, most of the built-in Android widgets are not really designed for this sort of simple subclassing, so you may be disappointed in how well this particular technique works.

Third, your custom `View` class might be a composite widget — akin to an activity's contents, complete with layout and such, but encapsulated in its own class. This allows you to create something more elaborate than you will just by tweaking resources. We will see this later in the chapter with `ColorMixer`.

Fourth, you might want to implement your own layout manager, if your GUI rules do not fit well with `RelativeLayout`, `TableLayout`, or other built-in containers. For example, you might want to create a layout manager that more closely mirrors the "box model" approach taken by XUL and Flex, or you might want to create one that mirrors Swing's `FlowLayout` (laying widgets out horizontally until there is no more room on the current row, then start a new row).

Finally, you might want to do something totally different, where you need to draw the widget yourself. For example, the `ColorMixer` widget uses `SeekBar` widgets to control the mix of red, blue, and green. But, you might create a `ColorWheel` widget that draws a spectrum gradient, detects touch events, and lets the user pick a color that way.

Some of these techniques are fairly simple; others are fairly complex. All share some common traits, such as widget-defined attributes, that we will see throughout the remainder of this chapter.

Colors, Mixed How You Like Them

The classic way for a user to pick a color in a GUI is to use a color wheel like this one:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 482: A color wheel from the API samples

There is even code to make one in the API samples.

However, a color wheel like that is difficult to manipulate on a touch screen, particularly a capacitive touchscreen designed for finger input. Fingers are great for gross touch events and lousy for selecting a particular color pixel.

Another approach is to use a mixer, with sliders to control the red, green, and blue values:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

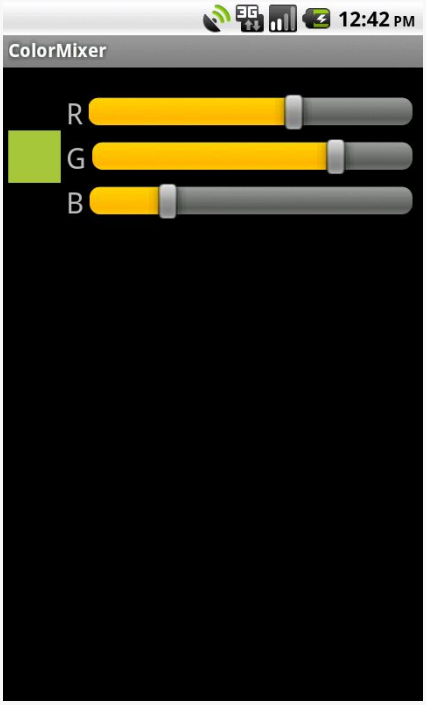


Figure 483: The ColorMixer widget, inside an activity

That is the custom widget you will see in this section, based on the code in the Views/ColorMixer sample project.

The Layout

`ColorMixer` is a composite widget, meaning that its contents are created from other widgets and containers. Hence, we can use a layout file to describe what the widget should look like.

The layout to be used for the widget is not that much: three `SeekBar` widgets (to control the colors), three `TextView` widgets (to label the colors), and one plain `View` (the "swatch" on the left that shows what the currently selected color is). Here is the file, found in `res/layout/mixer.xml` in the `Views/ColorMixer` project:

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android">
  <View android:id="@+id/swatch"
    android:layout_width="40dip"
    android:layout_height="40dip"
    android:layout_alignParentLeft="true"
    android:layout_centerVertical="true"
    android:layout_marginLeft="4dip"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  />
  <TextView android:id="@+id/redLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/swatch"
    android:layout_toRightOf="@id/swatch"
    android:layout_marginLeft="4dip"
    android:text="@string/red"
    android:textSize="10pt"
  />
  <SeekBar android:id="@+id/red"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/redLabel"
    android:layout_toRightOf="@id/redLabel"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="8dip"
  />
  <TextView android:id="@+id/greenLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/redLabel"
    android:layout_toRightOf="@id/swatch"
    android:layout_marginLeft="4dip"
    android:layout_marginTop="4dip"
    android:text="@string/green"
    android:textSize="10pt"
  />
  <SeekBar android:id="@+id/green"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/greenLabel"
    android:layout_toRightOf="@id/greenLabel"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="8dip"
  />
  <TextView android:id="@+id/blueLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/greenLabel"
    android:layout_toRightOf="@id/swatch"
    android:layout_marginLeft="4dip"
    android:layout_marginTop="4dip"
    android:text="@string/blue"
    android:textSize="10pt"
  />
  <SeekBar android:id="@+id/blue"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/blueLabel"
    android:layout_toRightOf="@id/blueLabel"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="8dip"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  />
</merge>
```

One thing that is a bit interesting about this layout, though, is the root element: `<merge>`. A `<merge>` layout is a bag of widgets that can be poured into some other container. The layout rules on the children of `<merge>` are then used in conjunction with whatever container they are added to. As we will see shortly, `ColorMixer` itself inherits from `RelativeLayout`, and the children of the `<merge>` element will become children of `ColorMixer` in Java. Basically, the `<merge>` element is only there because XML files need a single root — otherwise, the `<merge>` element itself is ignored in the layout.

The Attributes

Widgets usually have attributes that you can set in the XML file, such as the `android:src` attribute you can specify on an `ImageButton` widget. You can create your own custom attributes that can be used in your custom widget, by creating a `res/values/attrs.xml` file containing `declare-styleable` resources to specify them.

For example, here is the attributes file for `ColorMixer`:

```
<resources>
  <declare-styleable name="ColorMixer">
    <attr name="initialColor" format="color" />
  </declare-styleable>
</resources>
```

The `declare-styleable` element describes what attributes are available on the widget class specified in the name attribute — in our case, `ColorMixer`. Inside `declare-styleable` you can have one or more `attr` elements, each indicating the `name` of an attribute (e.g., `initialColor`) and what data `format` the attribute has (e.g., `color`). The data type will help with compile-time validation and in getting any supplied values for this attribute parsed into the appropriate type at runtime.

Here, we indicate there is only one attribute: `initialColor`, which will hold the initial color we want the mixer set to when it first appears.

There are many possible values for the `format` attribute in an `attr` element, including:

1. `boolean`
2. `color`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

3. `dimension`
4. `float`
5. `fraction`
6. `integer`
7. `reference` (which means a reference to another resource, such as a `Drawable`)
8. `string`

You can even support multiple formats for an attribute, by separating the values with a pipe (e.g., `reference|color`).

The Class

Our `ColorMixer` class, a subclass of `RelativeLayout`, will take those attributes and provide the actual custom widget implementation, for use in activities.

Constructor Flavors

A `View` has three possible constructors:

1. One takes just a `Context`, which usually will be an `Activity`
2. One takes a `Context` and an `AttributeSet`, the latter of which represents the attributes supplied via layout XML
3. One takes a `Context`, an `AttributeSet`, and the default style to apply to the attributes

If you are expecting to use your custom widget in layout XML files, you will need to implement the second constructor and chain to the superclass. If you want to use styles with your custom widget when declared in layout XML files, you will need to implement the third constructor and chain to the superclass. If you want developers to create instances of your `View` class in Java code directly, you probably should implement the first constructor and, again, chain to the superclass.

In the case of `ColorMixer`, all three constructors are implemented, eventually routing to the three-parameter edition, which initializes our widget. Below, you will see the first two of those constructors, with the third coming up in the next section:

```
public ColorMixer(Context context) {
  this(context, null);
}

public ColorMixer(Context context, AttributeSet attrs) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    this(context, attrs, 0);
  }
```

Using the Attributes

The `ColorMixer` has a starting color — after all, the `SeekBar` widgets and swatch `View` have to show something. Developers can, if they wish, set that color via a `setColor()` method:

```
  public void setColor(int color) {
    red.setProgress(Color.red(color));
    green.setProgress(Color.green(color));
    blue.setProgress(Color.blue(color));
    swatch.setBackgroundColor(color);
  }
```

If, however, we want developers to be able to use layout XML, we need to get the value of `initialColor` out of the supplied `AttributeSet`. In `ColorMixer`, this is handled in the three-parameter constructor:

```
  public ColorMixer(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);

    ((Activity)getContext())
        .getLayoutInflater()
        .inflate(R.layout.mixer, this, true);

    swatch=findViewById(R.id.swatch);

    red=(SeekBar)findViewById(R.id.red);
    red.setMax(0xFF);
    red.setOnSeekBarChangeListener(onMix);

    green=(SeekBar)findViewById(R.id.green);
    green.setMax(0xFF);
    green.setOnSeekBarChangeListener(onMix);

    blue=(SeekBar)findViewById(R.id.blue);
    blue.setMax(0xFF);
    blue.setOnSeekBarChangeListener(onMix);

    if (attrs!=null) {
      TypedArray a=getContext()
                      .obtainStyledAttributes(attrs,
                                              R.styleable.ColorMixer,
                                              0, 0);

      setColor(a.getInt(R.styleable.ColorMixer_initialColor,
                        0xFFA4C639));
      a.recycle();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }
```

There are three steps for getting attribute values:

- Get a `TypedArray` conversion of the `AttributeSet` by calling `obtainStyledAttributes()` on our `Context`, supplying it the `AttributeSet` and the ID of our styleable resource (in this case, `R.styleable.ColorMixer`, since we set the name of the `declare-styleable` element to be `ColorMixer`)
- Use the `TypedArray` to access specific attributes of interest, by calling an appropriate getter (e.g., `getInt()`) with the ID of the specific attribute to fetch (`R.styleable.ColorMixer_initialColor`)
- Recycle the `TypedArray` when done, via a call to `recycle()`, to make the object available to Android for use with other widgets via an object pool (versus creating new instances every time)

Note that the name of any given attribute, from the standpoint of `TypedArray`, is the name of the styleable resource (`R.styleable.ColorMixer`) concatenated with an underscore and the name of the attribute itself (`_initialColor`).

In `ColorMixer`, we get the attribute and pass it to `setColor()`. Since `getInt()` on `AttributeSet` takes a default value, we supply some stock color that will be used if the developer declined to supply an `initialColor` attribute.

Also note that our `ColorMixer` constructor inflates the widget's layout. In particular, it supplies `true` as the third parameter to `inflate()`, meaning that the contents of the layout should be added as children to the `ColorMixer` itself. When the layout is inflated, the `<merge>` element is ignored, and the `<merge>` element's children are added as children to the `ColorMixer`.

Saving the State

Similar to activities, a custom `View` overrides `onSaveInstanceState()` and `onRestoreInstanceState()` to persist data as needed, such as to handle a screen orientation change. The biggest difference is that rather than receive a `Bundle` as a parameter, `onSaveInstanceState()` must return a `Parcelable` with its state... including whatever state comes from the parent `View`.

The simplest way to do that is to return a `Bundle`, in which we have filled in our state (the chosen color) and the parent class' state (whatever that may be).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

So, for example, here are implementations of `onSaveInstanceState()` and `onRestoreInstanceState()` from `ColorMixer`:

```
  @Override
  public Parcelable onSaveInstanceState() {
    Bundle state=new Bundle();

    state.putParcelable(SUPERSTATE, super.onSaveInstanceState());
    state.putInt(COLOR, getColor());

    return(state);
  }

  @Override
  public void onRestoreInstanceState(Parcelable ss) {
    Bundle state=(Bundle)ss;

    super.onRestoreInstanceState(state.getParcelable(SUPERSTATE));

    setColor(state.getInt(COLOR));
  }
```

The Rest of the Functionality

`ColorMixer` defines a callback interface, named `OnColorChangedListener`:

```
  public interface OnColorChangedListener {
    public void onColorChange(int argb);
  }
```

`ColorMixer` also provides getters and setters for an `OnColorChangedListener` object:

```
  public OnColorChangedListener getOnColorChangedListener() {
    return(listener);
  }

  public void setOnColorChangedListener(OnColorChangedListener listener) {
    this.listener=listener;
  }
```

The rest of the logic is mostly tied up in the `SeekBar` handler, which will adjust the swatch based on the new color and invoke the `OnColorChangedListener` object, if there is one:

```
  private SeekBar.OnSeekBarChangeListener onMix=new
SeekBar.OnSeekBarChangeListener() {
    public void onProgressChanged(SeekBar seekBar, int progress,
                                  boolean fromUser) {
      int color=getColor();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      swatch.setBackgroundColor(color);

      if (listener!=null) {
        listener.onColorChange(color);
      }
    }

    public void onStartTrackingTouch(SeekBar seekBar) {
      // unused
    }

    public void onStopTrackingTouch(SeekBar seekBar) {
      // unused
    }
  };
```

Seeing It In Use

The project contains a sample activity, `ColorMixerDemo`, that shows the use of the `ColorMixer` widget.

The layout for that activity, shown below, can be found in `res/layout/main.xml` of the `Views/ColorMixer` project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:mixer="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical"
>
  <TextView android:id="@+id/color"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
  />
  <com.commonsware.android.colormixer.ColorMixer
    android:id="@+id/mixer"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    mixer:initialColor="#FFA4C639"
  />
</LinearLayout>
```

Notice that the root `LinearLayout` element defines two namespaces, the standard `android` namespace, and a separate one named `mixer`. The `mixer` namespace is given a URL of `http://schemas.android.com/apk/res-auto`, which indicates to the Android build system to match up `mixer` attributes with their respective widgets that are supplied via Android library projects.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Our `ColorMixer` widget is in the layout, with a fully-qualified class name (`com.commonsware.android.colormixer.ColorMixer`), since `ColorMixer` is not in the `android.widget` package. Notice that we can treat our custom widget like any other, giving it a width and height and so on.

The one attribute of our `ColorMixer` widget that is unusual is `mixer:initialColor`. `initialColor`, you may recall, was the name of the attribute we declared in `res/values/attrs.xml` and retrieve in Java code, to represent the color to start with. The `mixer` namespace is needed to identify where Android should be pulling the rules for what sort of values an `initialColor` attribute can hold. Since our `<attr>` element indicated that the `format` of `initialColor` was `color`, Android will expect to see a color value here, rather than a string or dimension.

The `ColorMixerDemo` activity is not very elaborate:

```
package com.commonsware.android.colormixer;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class ColorMixerDemo extends Activity {
  private TextView color=null;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    color=(TextView)findViewById(R.id.color);

    ColorMixer mixer=(ColorMixer)findViewById(R.id.mixer);

    mixer.setOnColorChangedListener(onColorChange);
  }

  private ColorMixer.OnColorChangedListener onColorChange=
    new ColorMixer.OnColorChangedListener() {
    public void onColorChange(int argb) {
      color.setText(Integer.toHexString(argb));
    }
  };
}
```

It gets access to both the `ColorMixer` and the `TextView` in the main layout, then registers an `OnColorChangedListener` with the `ColorMixer`. That listener, in turn, puts the value of the color in the `TextView`, so the user can see the hex value of the color along with the shade itself in the swatch.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

ReverseChronometer: Simply a Custom Subclass

Sometimes, what you want to achieve only requires a basic subclass of an existing widget (or container), into which you can pour your business logic.

For example, Android has a `Chronometer` widget, which is used for denoting elapsed time of some operation. It works well, but it only counts up from zero. It cannot be used to display a countdown instead.

But, we can roll a `ReverseChronometer` that does, simply by subclassing `TextView`, as seen in the Views/ReverseChronometer sample project:

```
package com.commonsware.android.revchron;

import android.content.Context;
import android.graphics.Color;
import android.os.SystemClock;
import android.util.AttributeSet;
import android.widget.TextView;

public class ReverseChronometer extends TextView implements Runnable {
  long startTime=0L;
  long overallDuration=0L;
  long warningDuration=0L;

  public ReverseChronometer(Context context, AttributeSet attrs) {
    super(context, attrs);

    reset();
  }

  @Override
  public void run() {
    long elapsedSeconds=
        (SystemClock.elapsedRealtime() - startTime) / 1000;

    if (elapsedSeconds < overallDuration) {
      long remainingSeconds=overallDuration - elapsedSeconds;
      long minutes=remainingSeconds / 60;
      long seconds=remainingSeconds - (60 * minutes);

      setText(String.format("%d:%02d", minutes, seconds));

      if (warningDuration > 0 && remainingSeconds < warningDuration) {
        setTextColor(0xFFFF6600); // orange
      }
      else {
        setTextColor(Color.BLACK);
      }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      postDelayed(this, 1000);
    }
    else {
      setText("0:00");
      setTextColor(Color.RED);
    }
  }

  public void reset() {
    startTime=SystemClock.elapsedRealtime();
    setText("--:--");
    setTextColor(Color.BLACK);
  }

  public void stop() {
    removeCallbacks(this);
  }

  public void setOverallDuration(long overallDuration) {
    this.overallDuration=overallDuration;
  }

  public void setWarningDuration(long warningDuration) {
    this.warningDuration=warningDuration;
  }
}
```

`ReverseChronometer` is designed to show minutes and seconds remaining from some initial time. In the constructor, by means to a call to a `reset()` method, we set the text of the `TextView` to show a generic starting point ("–:–"), set its color to black, and note the current time (`SystemClock.elapsedRealtime()`) in a `startTime` data member.

`ReverseChronometer` also tracks two durations in seconds, with corresponding setter methods:

- `overallDuration` is how long the countdown should run from beginning to end
- `warningDuration` is how far from the end we should change the color of the `TextView` from black to orange, to hint to the viewer that time is running out

`ReverseChronometer` implements `Runnable`, and when its `run()` method is called, it determines how many seconds have elapsed since that `startTime` value. Depending on the amount of seconds remaining, we either:

- Just update the text to show the minutes and seconds remaining
- Update the text and set the color to black or orange

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Set the text to "0:00" (time has run out) and set the text color to red

In either of the first two cases, we also call `postDelayed()` to schedule ourselves to run again in a second, where we can update the `TextView` contents once more. That continues until somebody calls `stop()`.

As with any custom `View`, we can reference this in a layout XML resource, fully-qualifying the class name used as the name of our XML element for the widget. And, since we inherit from `TextView`, we can set any of the attributes that we want on that `TextView`, in terms of styling the text, positioning it within a parent container, etc.:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".MainActivity">

  <com.commonsware.android.revchron.ReverseChronometer
    android:id="@+id/chrono"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:textSize="50sp"
    android:textStyle="bold"/>

</RelativeLayout>
```

All our activity needs to do is set the durations, then call `run()` and `stop()` at appropriate times, such as when the activity is resumed and paused:

```
package com.commonsware.android.revchron;

import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {
  private ReverseChronometer chrono=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    chrono=(ReverseChronometer)findViewById(R.id.chrono);
    chrono.setOverallDuration(90);
    chrono.setWarningDuration(10);
  }

  @Override
  public void onResume() {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onResume();

    chrono.run();
  }

  @Override
  public void onPause() {
    chrono.stop();

    super.onPause();
  }
}
```

The result is much as you would expect: a countdown of the time remaining:

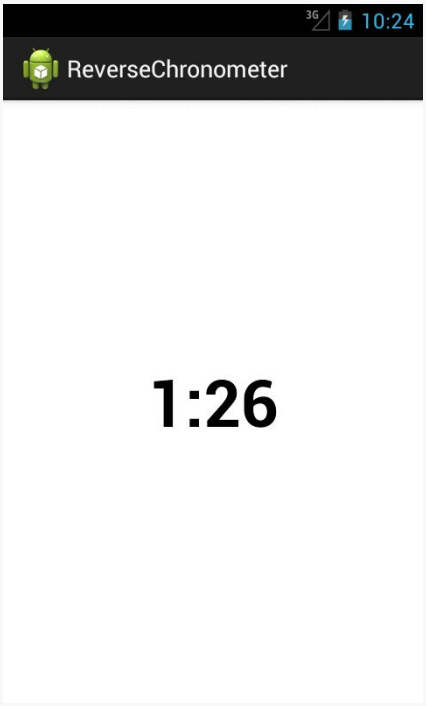


Figure 484: ReverseChronometer, Early in Countdown

...changing to orange when we are within the warning duration:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

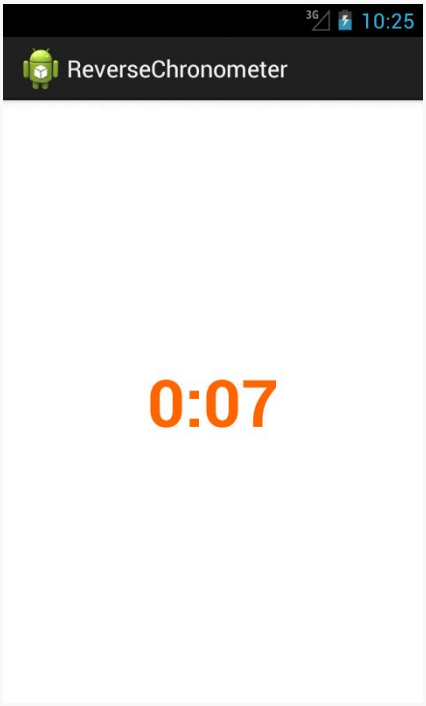


Figure 485: ReverseChronometer, Late in Countdown

...and changing to red when time has run out:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

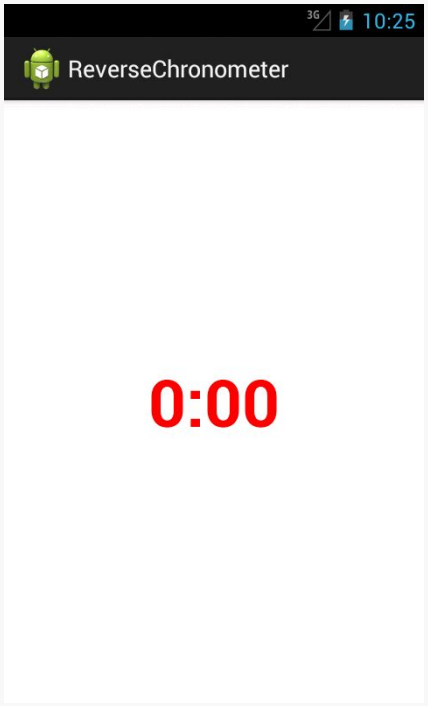


Figure 486: ReverseChronometer, With Complete Time Elapsed

Of course, much more could be done with this widget, if you chose:

- Support other constructors, beyond the two-argument constructor needed for layout inflation
- Support setting durations and colors via custom XML attributes
- Adding listeners for warning and expired events, so other things can be done at those points in time (e.g., play a sound, vibrate the device)

AspectLockedFrameLayout: A Custom Container

You can also craft your own custom container classes, whether inheriting straight from `ViewGroup` to implement your own set of layout rules, or by extending an existing `ViewGroup` to merely augment its functionality.

For example, there may be cases where you want to control the aspect ratio of some set of widgets. This is important when working with preview frames off of the `Camera` to prevent distortion, for example.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`AspectLockedFrameLayout`, therefore, is a custom extension of `FrameLayout` that ensures that its contents are kept within a particular aspect ratio, reducing the height or width of the contents to keep that aspect ratio.

`AspectLockedFrameLayout` is published as part of the CWAC-Layouts project, with its own GitHub repo. As with many of the CWAC projects, the reusable code is distributed as a JAR and as an Android library project, with a `demo/` sub-project illustrating the use of some of the library's contents.

`AspectLockedFrameLayout` holds onto two data members:

- A `double` (`aspectRatio`) that represents a specific aspect ratio to maintain, initialized to `0.0`
- A `View` (`aspectRatioSource`) that represents some other widget whose aspect ratio should be matched, initialized to `null`

`AspectLockedFrameLayout` has corresponding setters for each:

```
public void setAspectRatioSource(View v) {
  this.aspectRatioSource=new ViewAspectRatioSource(v);
}

public void setAspectRatioSource(AspectRatioSource aspectRatioSource) {
  this.aspectRatioSource=aspectRatioSource;
}

// from com.android.camera.PreviewFrameLayout, with slight
// modifications

public void setAspectRatio(double aspectRatio) {
  if (aspectRatio <= 0.0) {
    throw new IllegalArgumentException(
                                       "aspect ratio must be positive");
  }

  if (this.aspectRatio != aspectRatio) {
    this.aspectRatio=aspectRatio;
    requestLayout();
  }
}
```

The "business logic" of maintaining the aspect ratio comes in `onMeasure()`. `onMeasure()` is called on a `ViewGroup` when it is time for it to determine its actual size, based upon things like the requested height and width and the sizes of its children. In our case `onMeasure()` needs to be tweaked to maintain the aspect ratio, assuming that we have an aspect ratio to work with:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  // from com.android.camera.PreviewFrameLayout, with slight
  // modifications

  @Override
  protected void onMeasure(int widthSpec, int heightSpec) {
    double localRatio=aspectRatio;

    if (localRatio == 0.0 && aspectRatioSource != null
        && aspectRatioSource.getHeight() > 0) {
      localRatio=
          (double)aspectRatioSource.getWidth()
              / (double)aspectRatioSource.getHeight();
    }

    if (localRatio == 0.0) {
      super.onMeasure(widthSpec, heightSpec);
    }
    else {
      int lockedWidth=MeasureSpec.getSize(widthSpec);
      int lockedHeight=MeasureSpec.getSize(heightSpec);

      if (lockedWidth == 0 && lockedHeight == 0) {
        throw new IllegalArgumentException(
                                          "Both width and height cannot be zero
-- watch out for scrollable containers");
      }

      // Get the padding of the border background.
      int hPadding=getPaddingLeft() + getPaddingRight();
      int vPadding=getPaddingTop() + getPaddingBottom();

      // Resize the preview frame with correct aspect ratio.
      lockedWidth-=hPadding;
      lockedHeight-=vPadding;

      if (lockedHeight > 0 && (lockedWidth > lockedHeight * localRatio)) {
        lockedWidth=(int)(lockedHeight * localRatio + .5);
      }
      else {
        lockedHeight=(int)(lockedWidth / localRatio + .5);
      }

      // Add the padding of the border.
      lockedWidth+=hPadding;
      lockedHeight+=vPadding;

      // Ask children to follow the new preview dimension.
      super.onMeasure(MeasureSpec.makeMeasureSpec(lockedWidth,
                                                  MeasureSpec.EXACTLY),
                      MeasureSpec.makeMeasureSpec(lockedHeight,
                                                  MeasureSpec.EXACTLY));
    }
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We start by determining what actually is the desired aspect ratio, held onto in a `localRatio` local variable. That will be `aspectRatio` if we do not have an `aspectRatioSource` that already knows its size, otherwise we will calculate the aspect ratio from the source. And, if `localRatio` turns out to be `0.0`, indicating that we do not have an aspect ratio to maintain, we just chain to the superclass, so `AspectLockedFrameLayout` will behave just like a normal `FrameLayout`.

If we *do* have an aspect ratio to maintain, we start by determining our requested height and width. `onMeasure()` is passed a pair of "specs" that provides details about our requested size, and we can get the height and width from those by means of the `MeasureSpec` helper class. We remove any horizontal padding — padding is considered to be "outside" the locked area and therefore is ignored in aspect ratio calculations. We then adjust the height or the width, as needed, to maintain the aspect ratio. We add back in the padding, then chain to the superclass with revised height and width "specs" via `MeasureSpec`.

Note that much of this logic was derived from `com.android.camera.PreviewFrameLayout` from the AOSP Camera application, which is used to maintain the aspect ratio of the `SurfaceView` used to display preview frames.

To use an `AspectLockedFrameLayout`, just add it to your layout XML file, with an appropriate child widget/container representing the material that needs to maintain a particular aspect ratio. Since the `AspectLockedFrameLayout` is overriding its natural size, you can use `android:layout_gravity` to control its positioning within some parent widget, such as centering it:

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.commonsware.cwac.layouts.AspectLockedFrameLayout
        android:id="@+id/source"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="center">

        <!-- children go here -->
    </com.commonsware.cwac.layouts.AspectLockedFrameLayout>
</FrameLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Mirror and MirroringFrameLayout: Draw It Yourself

Another scenario where aspect ratios matter is when you are presenting information on an external display via `Presentation`, as is covered elsewhere in this book. Ideally, you fill the external display. And normally this will happen for you automatically, as your `Presentation` content view should fill the available screen space... assuming that the content has the right aspect ratio, or can be suitably stretched.

One scenario where this might be a problem is if you want *the same material* shown on both the main display and on the external display. For example, suppose that you are using `Presentation` to deliver... well... a presentation. The external display is probably some form of video projector, and you will want your slides or other materials shown there. However, it is useful for you to be able to see those same slides and such on the tablet, as typically the projector screen is behind, or to the side of, the presenter. If the presenter has to keep turning around to confirm what is shown on "the big screen", it can detract from the presentation.

Moreover, you might not only want to *show* the same material, but have it stem *from the same source*, on the tablet, for interactivity reasons. Suppose that you want to display a Web page. You might just pop up a `WebView` in the `Presentation`. But... how do you scroll? The `Presentation` offers no touch interface — projector screens do not magically respond to pinch-to-zoom just because we happen to be projecting something onto them from an Android tablet.

In this case, ideally we would like to *mirror* something. Have the actual widgets shown on the tablet, which can then respond to touch events and the like. At the same time, capture what is shown on the tablet and reproduce it, verbatim, on the `Presentation` for the audience to see. Now everybody can see the same material, and the presenter can manipulate that material.

But now aspect ratios come into play. We want to fill the `Presentation` display space, without black bars or stretching or whatever. That only works if our source material — the widgets and containers to be mirrored — have the same aspect ratio as the `Presentation`'s `Display` itself.

With that in mind, the CWAC Layouts project also contains two classes to solve this problem:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `MirroringFrameLayout` is an `AspectLockedFrameLayout` that also can mirror its content to...
- `Mirror`, a `View` that takes a `Bitmap` representing the `MirroringFrameLayout` contents and displays it

Technically, `MirroringFrameLayout` works with a `MirrorSink`, an interface that can receive updates to the content to be mirrored when that content changes. `Mirror` implements `MirrorSink`, and you could have other classes implement `MirrorSink` as well if that made sense for your app. The sections that follow focus on `MirroringFrameLayout` working with a `Mirror`, as that is the most likely scenario.

MirroringFrameLayout

`MirroringFrameLayout` extends `AspectLockedFrameLayout`, so that we can lock the aspect ratio of the to-be-mirrored contents to match the aspect ratio of the `Mirror`. The `Mirror` is designed to be projected by the `Presentation`, and so if the `Mirror` fills the `Presentation`'s `Display`, we want our `MirroringFrameLayout` to match the aspect ratio so the entire `Display` can indeed be filled.

Of course, a `ViewGroup` like `FrameLayout` normally just has its children draw to the screen. In our case, we need to capture what is drawn ourselves, to supply to the `Mirror` as needed. This is a bit tricky.

```
package com.commonsware.cwac.layouts;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Rect;
import android.util.AttributeSet;
import android.view.ViewTreeObserver.OnPreDrawListener;
import android.view.ViewTreeObserver.OnScrollChangedListener;

public class MirroringFrameLayout extends AspectLockedFrameLayout
    implements OnPreDrawListener, OnScrollChangedListener {
  private MirrorSink mirror=null;
  private Bitmap bmp=null;
  private Canvas bmpBackedCanvas=null;
  private Rect rect=new Rect();

  public MirroringFrameLayout(Context context) {
    this(context, null);
  }

  public MirroringFrameLayout(Context context, AttributeSet attrs) {
    super(context, attrs);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    setWillNotDraw(false);
  }

  public void setMirror(MirrorSink mirror) {
    this.mirror=mirror;

    if (mirror != null) {
      setAspectRatioSource(mirror);
    }
  }

  @Override
  public void onAttachedToWindow() {
    super.onAttachedToWindow();

    getViewTreeObserver().addOnPreDrawListener(MirroringFrameLayout.this);
    getViewTreeObserver().addOnScrollChangedListener(MirroringFrameLayout.this);
  }

  @Override
  public void onDetachedFromWindow() {
    getViewTreeObserver().removeOnPreDrawListener(this);
    getViewTreeObserver().removeOnScrollChangedListener(this);

    super.onDetachedFromWindow();
  }

  @Override
  public void draw(Canvas canvas) {
    if (mirror != null) {
      bmp.eraseColor(0);

      super.draw(bmpBackedCanvas);
      getDrawingRect(rect);
      canvas.drawBitmap(bmp, null, rect, null);
      mirror.update(bmp);
    }
    else {
      super.draw(canvas);
    }
  }

  @Override
  protected void onSizeChanged(int w, int h, int oldw, int oldh) {
    initBitmap(w, h);

    super.onSizeChanged(w, h, oldw, oldh);
  }

  @Override
  public boolean onPreDraw() {
    if (mirror != null) {
      if (bmp == null) {
        requestLayout();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      }
      else {
        invalidate();
      }
    }

    return(true);
  }

  @Override
  public void onScrollChanged() {
    onPreDraw();
  }

  private void initBitmap(int w, int h) {
    if (mirror != null) {
      if (bmp == null || bmp.getWidth() != w || bmp.getHeight() != h) {
        if (bmp != null) {
          bmp.recycle();
        }

        bmp=Bitmap.createBitmap(w, h, Bitmap.Config.ARGB_8888);
        bmpBackedCanvas=new Canvas(bmp);
      }
    }
  }
}
```

Our one-argument constructor uses `this()` to chain to the two-argument constructor. The two-argument constructor calls `setWillNotDraw(false)` indicating to Android that we want this `ViewGroup` to participate in the drawing process like a regular `View` — normally, certain steps in the drawing process are skipped as being irrelevant to `View` classes that do not draw anything themselves.

We have a `setMirror()` method, where the activity or fragment can supply the `MirrorSink` that is connected to this `MirroringFrameLayout`. In addition to holding onto the `MirrorSink` in a `mirror` data member, we call `setAspectRatioSource()`, inherited from `AspectLockedFrameLayout`, so our contents will match the aspect ratio from that source.

`MirroringFrameLayout` overrides `onAttachedToWindow()` and `onDetatchedFromWindow()`. As one might guess, these callbacks are called when views are attached and detached from some window. Usually, that window represents an activity, though it could represent a `Dialog` or a `Presentation`.

In those callbacks, we connect with the `ViewTreeObserver` of the `MirroringFrameLayout`. A `ViewTreeObserver` is a way to find out about events of a view tree, rooted at some `ViewGroup`. In our case, we want to find out when children

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

are going to be drawn (`addOnPreDrawListener()`) and when they are scrolled (`addOnScrollChangedListener()`).

We override `onSizeChanged()`. This is called on any `View` when its size may have changed, either because it is being sized initially when the UI is being set up, or because something else nearby changed size (e.g., its parent) and therefore the size of the `View` itself may now be different. In our case, we use `onSizeChanged()` to set up a `Bitmap` object, sized to match our size, and a `Canvas` object that wraps around that `Bitmap` object. As you will see, we will use this `Canvas` to capture what is being drawn on the screen, for later use by the `Mirror`.

We also override `draw()`. This is, in effect, the "entry point" into the logic that causes a `View` to render itself on the screen, by drawing to a supplied `Canvas` object. Most `View` classes do not override `draw()`, as the real rendering is done in an `onDraw()` method, as we will see with `Mirror` later in this chapter. However, in our case, we have to override `draw()` for one simple reason: we do not want to draw to the `Canvas` supplied by Android to the `draw()` method. We want to draw to our own `Canvas`, backed by that `Bitmap`.

To that end, if we have a `MirrorSink`, we:

- Make sure the `Bitmap` starts off blank by calling `eraseColor()`
- Chain to the superclass, replacing the `Canvas` given to us in `draw()` by our own `Bitmap`-backed `Canvas`
- Calculate a `Rect` object with our size and position, using `getDrawingRect()`
- Use that `Rect` and the `Bitmap` to render the `Bitmap` to the "real" `Canvas` supplied to us in `draw()`
- Call `update()` on the `MirrorSink`, to give it the new `Bitmap`

By rendering our contents to the `Bitmap`-backed `Canvas`, instead of the normal one, we capture a copy of the output, in the form of the `Bitmap`. Since the `Bitmap` has the same size as the "real" `Canvas` (courtesy of our `onSizeChanged()` work), when we draw the `Bitmap` onto the `Canvas`, we effectively "color in" the same pixels in the same spots as if we had skipped all of this and left the normal `draw()` logic alone. But, since we still hold onto our `Bitmap`, we can use those same pixels elsewhere... such as in our `Mirror`.

The problem with relying on `draw()` is that it is not always called when there are changes to widgets within the `MirroringFrameLayout`. In particular, `WebView` often does not trigger `draw()` on the `MirroringFrameLayout`. That's where the pre-draw

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

and scroll-changed events from the `ViewTreeObserver` come into play: they give us more indication that we need to update our `Bitmap`.

The `onPreDraw()` method is called when a child of this `MirroringFrameLayout` is about to be drawn. If we have our `MirrorSink`, we then either call `requestLayout()` (if we have no bitmap yet) or `invalidate()` (if we do), to trigger Android to go through the draw process for the `MirroringFrameLayout` too, allowing us to update our `Bitmap`.

The `onScrollChanged()` method is called when a child of this `MirroringFrameLayout` has been scrolled. This delegates to `onPreDraw()`, to run through the same logic to force an update to the `Bitmap`.

Mirror

`Mirror` extends the base `View` class, and so it is the most "raw" of all the custom widgets and containers shown so far in this chapter. It has an `update()` method, used to connect the `MirroringFrameLayout` from which the `Mirror` can obtain what it is supposed to display:

```
package com.commonsware.cwac.layouts;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Rect;
import android.util.AttributeSet;
import android.view.View;

public class Mirror extends View implements MirrorSink {
  private Rect rect=new Rect();
  private Bitmap bmp=null;

  public Mirror(Context context) {
    super(context);
  }

  public Mirror(Context context, AttributeSet attrs) {
    super(context, attrs);
  }

  public Mirror(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
  }

  @Override
  public void update(Bitmap bmp) {
    this.bmp=bmp;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    invalidate();
  }

  @Override
  protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    if (bmp != null) {
      getDrawingRect(rect);

      calcCenter(rect.width(), rect.height(), bmp.getWidth(),
                 bmp.getHeight(), rect);
      canvas.drawBitmap(bmp, null, rect, null);
    }
  }

  // based upon http://stackoverflow.com/a/14679729/115145

  static void calcCenter(int vw, int vh, int iw, int ih, Rect out) {
    double scale=
        Math.min((double)vw / (double)iw, (double)vh / (double)ih);

    int h=(int)(scale * ih);
    int w=(int)(scale * iw);
    int x=((vw - w) >> 1);
    int y=((vh - h) >> 1);

    out.set(x, y, x + w, y + h);
  }
}
```

The bulk of the "business logic" lies in `onDraw()`, plus a helper `calcCenter()` static method.

`onDraw()` is called on a `View` when it is time for that widget to actually draw its visual representation onto the supplied `Canvas`. Different widgets will use different drawing primitive methods offered by `Canvas`, to draw lines and text and whatnot. In our case, we:

- Calculate a `Rect` object with our size and position, using `getDrawingRect()`
- Get the `Bitmap` object from the `MirroringFrameLayout`, via a call to `getLastBitmap()` (which simply returns the `Bitmap` that the `MirroringFrameLayout` is using)
- Call `calcCenter` to adjust our `Rect` to take into account the fact that our size may be different than the size of the actual `Bitmap`
- Call `drawBitmap()` on our `Canvas`, to render the `Bitmap` into the location specified by the `Rect`, where `drawBitmap()` will automatically down-sample or up-sample the image as needed to fill the necessary space

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Usage and Results

Normally, you would use the `Mirror` in a layout for a `Presentation` and the `MirroringFrameLayout` in an activity that controls the `Presentation`. However, it is possible to use both in the same layout file, for light testing. However, please do not put the `Mirror` inside of the `MirroringFrameLayout`, as this is likely to cause a rupture in the space-time continuum, and you really do not want to be responsible for that.

So, in the `SimpleMirrorActivity` from the `demo/` sub-project, we use a layout that has both `Mirror` and `MirroringFrameLayout`, with the latter set to mirror a `WebView`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  tools:context=".SimpleMirrorActivity">

  <FrameLayout
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1">

    <com.commonsware.cwac.layouts.MirroringFrameLayout
      android:id="@+id/source"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:layout_gravity="center">

      <EditText
        android:id="@+id/editor"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="left|top"
        android:inputType="textMultiLine"/>
    </com.commonsware.cwac.layouts.MirroringFrameLayout>
  </FrameLayout>

  <View
    android:layout_width="match_parent"
    android:layout_height="4dip"
    android:background="#FF000000"/>

  <com.commonsware.cwac.layouts.Mirror
    android:id="@+id/target"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="2"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
</LinearLayout>
```

In this case, we set the background of the `FrameLayout` holding our `MirroringFrameLayout` to green, to show how the `MirroringFrameLayout` size is changed to maintain our aspect ratio.

(or, perhaps we just like green)

Besides configuring the to-be-mirrored widgets, all you need to do is call `setMirror()` on the `MirroringFrameLayout` to enable the mirroring logic:

```
package com.commonsware.cwac.layouts.demo;

import android.app.Activity;
import android.os.Bundle;
import com.commonsware.cwac.layouts.Mirror;
import com.commonsware.cwac.layouts.MirroringFrameLayout;

public class SimpleMirrorActivity extends Activity {
  MirroringFrameLayout source=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.simple_mirror);

    source=(MirroringFrameLayout)findViewById(R.id.source);
    Mirror target=(Mirror)findViewById(R.id.target);

    source.setMirror(target);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

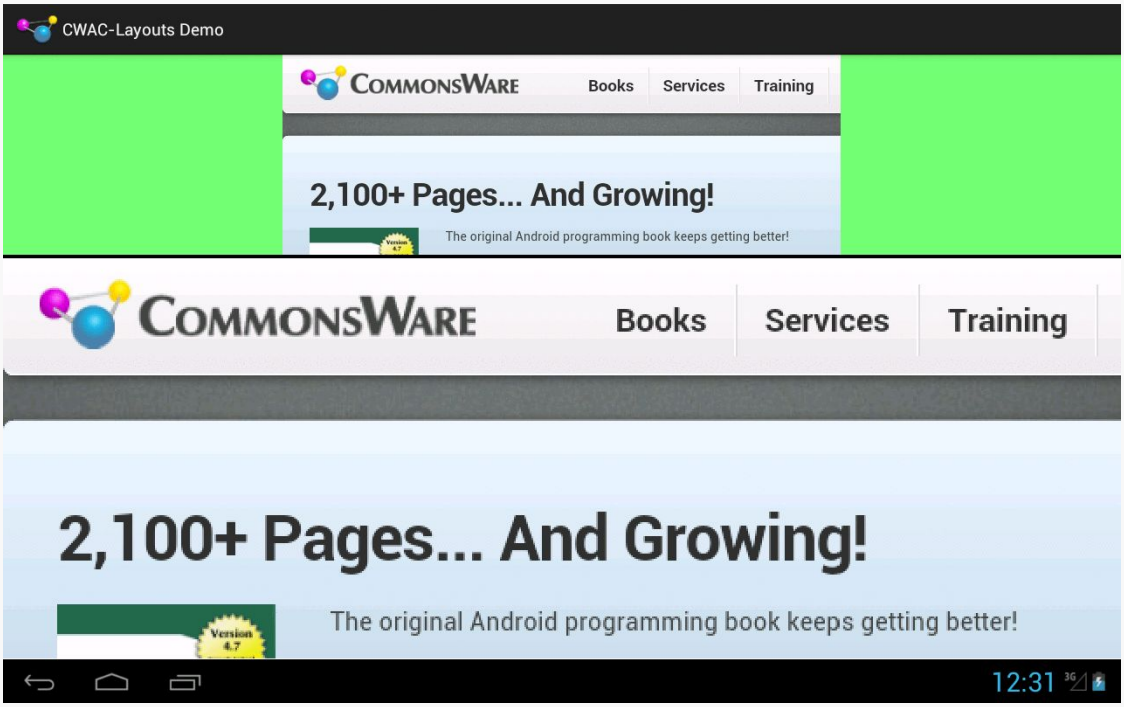


Figure 487: MirroringFrameLayout Above Its Mirror

While the bottom portion is just the `Mirror` and therefore is non-interactive, the top is the real `WebView`, which can be scrolled, with the resulting changes reflected in the `Mirror` in real-time:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

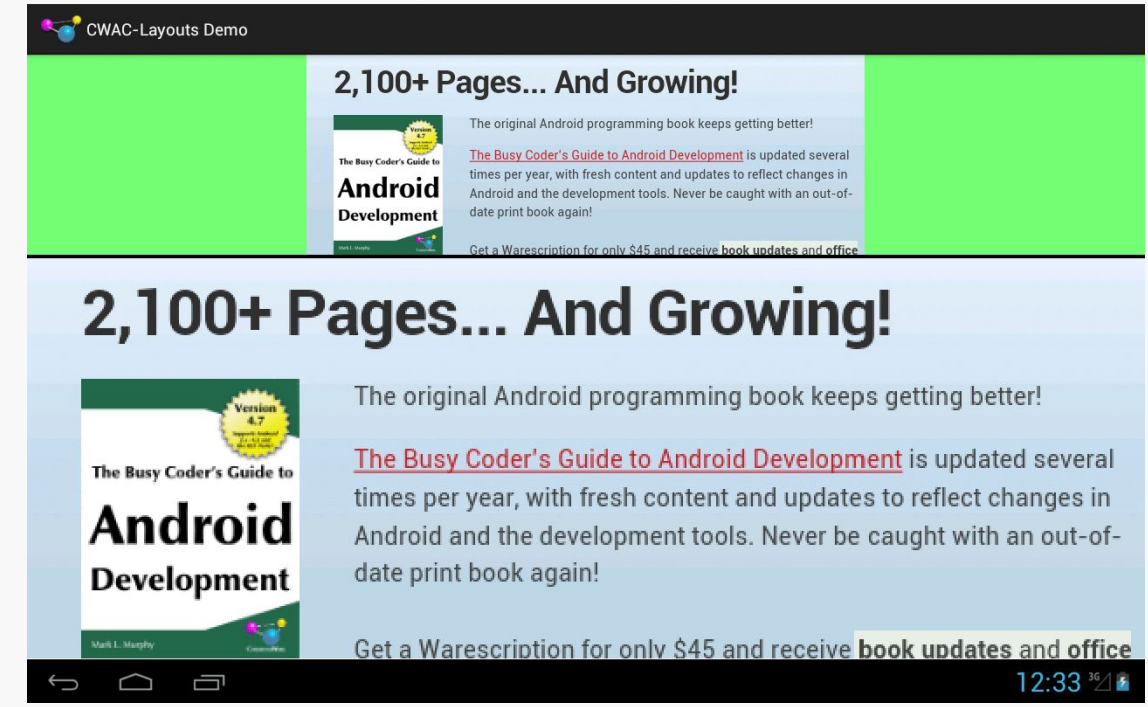


Figure 488: MirroringFrameLayout and Mirror, Showing Scrolled Contents

Limitations

`MirroringFrameLayout` only works for materials drawn in the Java layer, that therefore can be drawn to the `Bitmap`-backed `Canvas`. Content *not* drawn in the Java layer will not work with `MirroringFrameLayout`, notably anything involving a `SurfaceView`. This not only includes your own `SurfaceView` widgets, but anything else that depends upon `SurfaceView`, such as `VideoView` or the Maps V2 `MapView` and `MapFragment`.

Also, the re-sampling done by `Mirror` is not especially sophisticated and will cause jagged effects, particularly when up-sampling. Ideally, the `MirroredFrameLayout` will be the same size or larger than the `Mirror`. This may not always be possible, particularly with a `Mirror` shown on a 1080p external display, but the closer you can get will improve the output.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Custom Dialogs and Preferences

Android ships with a number of dialog classes for specific circumstances, like `DatePickerDialog` and `ProgressDialog`. Similarly, Android comes with a smattering of `Preference` classes for your `PreferenceActivity`, to accept text or selections from lists and so on.

However, there is plenty of room for improvement in both areas. As such, you may find the need to create your own custom dialog or preference class. This chapter will show you how that is done.

We start off by looking at creating a custom `AlertDialog`, not by using `AlertDialog.Builder`, but via a custom subclass. Then, we show how to create your own dialog-style `Preference`, where tapping on the preference pops up a dialog to allow the user to customize the preference value.

Prerequisites

Understanding this chapter requires that you have read the chapter on dialogs, along with the chapter on the preference system. Also, the samples here use the custom `ColorMixer View` described in another chapter.

Your Dialog, Chocolate-Covered

For your own application, the simplest way to create a custom `AlertDialog` is to use `AlertDialog.Builder`, as described in a previous chapter. You do not need to create any special subclass — just call methods on the `Builder`, then `show()` the resulting dialog.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, if you want to create a reusable `AlertDialog`, this may become problematic. For example, where would this code to create the custom `AlertDialog` reside?

So, in some cases, you may wish to extend `AlertDialog` and supply the dialog's contents that way, which is how `TimePickerDialog` and others are implemented. Unfortunately, this technique is not well documented. This section will illustrate how to create such an `AlertDialog` subclass, as determined by looking at how the core Android team did it for their own dialogs.

The sample code is `ColorMixerDialog`, a dialog wrapping around the `ColorMixer` widget shown in a previous chapter. The implementation of `ColorMixerDialog` can be found in the CWAC-ColorMixer GitHub repository, as it is part of the CommonsWare Android Components.

Using this dialog works much like using `DatePickerDialog` or `TimePickerDialog`. You create an instance of `ColorMixerDialog`, supplying the initial color to show and a listener object to be notified of color changes. Then, call `show()` on the dialog. If the user makes a change and accepts the dialog, your listener will be informed.

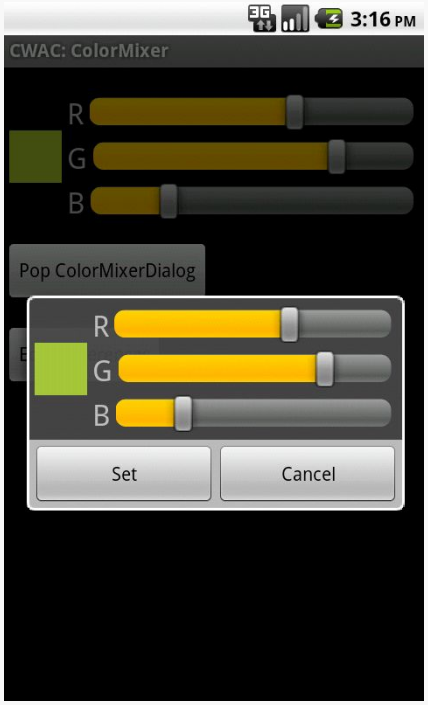


Figure 489: The ColorMixerDialog

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Basic AlertDialog Setup

The `ColorMixerDialog` class is not especially long, since all of the actual color mixing is handled by the `ColorMixer` widget:

```
package com.commonsware.cwac.colormixer;

import android.app.AlertDialog;
import android.content.Context;
import android.content.DialogInterface;
import android.os.Bundle;

public class ColorMixerDialog extends AlertDialog
  implements DialogInterface.OnClickListener {
  static private final String COLOR="c";
  private ColorMixer mixer=null;
  private int initialColor;
  private ColorMixer.OnColorChangedListener onSet=null;

  public ColorMixerDialog(Context ctxt,
                          int initialColor,
                          ColorMixer.OnColorChangedListener onSet) {
    super(ctxt);

    this.initialColor=initialColor;
    this.onSet=onSet;

    mixer=new ColorMixer(ctxt);
    mixer.setColor(initialColor);

    setView(mixer);
    setButton(ctxt.getText(R.string.cwac_colormixer_set),
              this);
    setButton2(ctxt.getText(R.string.cwac_colormixer_cancel),
               (DialogInterface.OnClickListener)null);
  }

  @Override
  public void onClick(DialogInterface dialog, int which) {
    if (initialColor!=mixer.getColor()) {
      onSet.onColorChange(mixer.getColor());
    }
  }

  @Override
  public Bundle onSaveInstanceState() {
    Bundle state=super.onSaveInstanceState();

    state.putInt(COLOR, mixer.getColor());

    return(state);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void onRestoreInstanceState(Bundle state) {
    super.onRestoreInstanceState(state);

    mixer.setColor(state.getInt(COLOR));
  }
}
```

We extend the `AlertDialog` class and implement a constructor of our own design. In this case, we take in three parameters:

1. A `Context` (typically an `Activity`), needed for the superclass
2. The initial color to use for the dialog, such as if the user is editing a color they chose before
3. A `ColorMixer.OnColorChangedListener` object, just like `ColorMixer` uses, to notify the dialog creator when the color is changed

We then create a `ColorMixer` and call `setView()` to make that be the main content of the dialog. We also call `setButton()` and `setButton2()` to specify a "Set" and "Cancel" button for the dialog. The latter just dismisses the dialog, so we need no event handler. The former we route back to the `ColorMixerDialog` itself, which implements the `DialogInterface.OnClickListener` interface.

Handling Color Changes

When the user clicks the "Set" button, we want to notify the application about the color change...if the color actually changed. This is akin to `DatePickerDialog` and `TimePickerDialog` only notifying you of date or times if the user clicks Set and actually changed the values.

The `ColorMixerDialog` tracks the initial color via the `initialColor` data member. In the `onClick()` method — required by `DialogInterface.OnClickListener` — we see if the mixer has a different color than the `initialColor`, and if so, we call the supplied `ColorMixer.OnColorChangedListener` callback object:

```
  @Override
  public void onClick(DialogInterface dialog, int which) {
    if (initialColor!=mixer.getColor()) {
      onSet.onColorChange(mixer.getColor());
    }
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

State Management

Dialogs use `onSaveInstanceState()` and `onRestoreInstanceState()`, just like activities do. That way, if the screen is rotated, or if the hosting activity is being evicted from RAM when it is not in the foreground, the dialog can save its state, then get it back later as needed.

The biggest difference with `onSaveInstanceState()` for a dialog is that the `Bundle` of state data is not passed into the method. Rather, you get the `Bundle` by chaining to the superclass, then adding your data to the `Bundle` it returned, before returning it yourself:

```
@Override
public Bundle onSaveInstanceState() {
  Bundle state=super.onSaveInstanceState();

  state.putInt(COLOR, mixer.getColor());

  return(state);
}
```

The `onRestoreInstanceState()` pattern is much closer to the implementation you would find in an `Activity`, where the `Bundle` with the state data to restore is passed in as a parameter:

```
@Override
public void onRestoreInstanceState(Bundle state) {
  super.onRestoreInstanceState(state);

  mixer.setColor(state.getInt(COLOR));
}
```

Preferring Your Own Preferences, Preferably

The Android Settings application, built using the `Preference` system, has lots of custom `Preference` classes. You too can create your own `Preference` classes, to collect things like dates, numbers, or colors. Once again, though, the process of creating such classes is not well documented. This section reviews one recipe for making a `Preference` — specifically, a subclass of `DialogPreference` – based on the implementation of other `Preference` classes in Android.

The result is `ColorPreference`, a `Preference` that uses the `ColorMixer` widget. As with the `ColorMixerDialog` from the previous section, the `ColorPreference` is from

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the CommonsWare Android Components, and its source code can be found in the CWAC-ColorMixer GitHub repository.

One might think that `ColorPreference`, as a subclass of `DialogPreference`, might use `ColorMixerDialog`. However, that is not the way it works, as you will see.

The Constructor

A Preference is much like a custom `View`, in that there are a variety of constructors, some taking an `AttributeSet` (for the preference properties), and some taking a default style. In the case of `ColorPreference`, we need to get the string resources to use for the names of the buttons in the dialog box, providing them to `DialogPreference` via `setPositiveButtonText()` and `setNegativeButtonText()`.

Here, we just implement the standard two-parameter constructor, since that is the one that is used when this preference is inflated from a preference XML file:

```
public ColorPreference(Context ctxt, AttributeSet attrs) {
  super(ctxt, attrs);

  setPositiveButtonText(ctxt.getText(R.string.cwac_colormixer_set));
  setNegativeButtonText(ctxt.getText(R.string.cwac_colormixer_cancel));
}
```

Creating the View

The `DialogPreference` class handles the pop-up dialog that appears when the preference is clicked upon by the user. Subclasses get to provide the `View` that goes inside the dialog. This is handled a bit reminiscent of a `CursorAdapter`, in that there are two separate methods to be overridden:

- `onCreateDialogView()` works like `newView()` of `CursorAdapter`, returning a `View` that should go in the dialog
- `onBindDialogView()` works like `bindView()` of `CursorAdapter`, where the custom `Preference` is supposed to configure the `View` for the current preference value

In the case of `ColorPreference`, we use a `ColorMixer` for the `View`:

```
@Override
protected View onCreateDialogView() {
  mixer=new ColorMixer(getContext());
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(mixer);
  }
```

Then, in `onBindDialogView()`, we set the mixer's color to be `lastColor`, a private data member:

```
  @Override
  protected void onBindDialogView(View v) {
    super.onBindDialogView(v);

    mixer.setColor(lastColor);
  }
```

We will see later in this section where `lastColor` comes from – for the moment, take it on faith that it holds the user's chosen color, or a default value.

Dealing with Preference Values

Of course, the whole point behind a `Preference` is to allow the user to set some value that the application will then use later on. Dealing with values is a bit tricky with `DialogPreference`, but not too bad.

Getting the Default Value

The preference XML format has an `android:defaultValue` attribute, which holds the default value to be used by the preference. Of course, the actual data type of the value will differ widely — an `EditTextPreference` might expect a `String`, while `ColorPreference` needs a color value.

Hence, you need to implement `onGetDefaultValue()`. This is passed a `TypedArray` — similar to how a custom `View` uses a `TypedArray` for getting at its custom attributes in an XML layout file. It is also passed an index number into the array representing `android:defaultValue`. The custom `Preference` needs to return an `Object` representing its interpretation of the default value.

In the case of `ColorPreference`, we simply get an integer out of the `TypedArray`, representing the color value, with an overall default value of `0xFFA4C639` (a.k.a., Android green):

```
  @Override
  protected Object onGetDefaultValue(TypedArray a, int index) {
    return(a.getInt(index, 0xFFA4C639));
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Setting the Initial Value

When the user clicks on the preference, the `DialogPreference` supplies the last-known preference value to its subclass, or the default value if this preference has not been set by the user to date.

The way this works is that the custom `Preference` needs to override `onSetInitialValue()`. This is passed in a boolean flag (`restoreValue`) indicating whether or not the user set the value of the preference before. It is also passed the `Object` returned by `onGetDefaultValue()`. Typically, a custom `Preference` will look at the flag and choose to either use the default value or load the already-set preference value.

To get the existing value, `Preference` defines a set of type-specific getter methods — `getPersistedInt()`, `getPersistedString()`, etc. So, `ColorPreference` uses `getPersistedInt()` to get the saved color value:

```
  @Override
  protected void onSetInitialValue(boolean restoreValue, Object defaultValue) {
    lastColor=(restoreValue ? getPersistedInt(lastColor) :
(Integer)defaultValue);
  }
```

Here, `onSetInitialValue()` stores that value in `lastColor` — which then winds up being used by `onBindDialogView()` to tell the `ColorMixer` what color to show.

Closing the Dialog

When the user closes the dialog, it is time to persist the chosen color from the `ColorMixer`. This is handled by the `onDialogClosed()` callback method on your custom `Preference`:

```
  @Override
  protected void onDialogClosed(boolean positiveResult) {
    super.onDialogClosed(positiveResult);

    if (positiveResult) {
      if (callChangeListener(mixer.getColor())) {
        lastColor=mixer.getColor();
        persistInt(lastColor);
      }
    }
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The passed-in boolean indicates if the user accepted or dismissed the dialog, so you can elect to skip saving anything if the user dismissed the dialog. The other `DialogPreference` implementations also call `callChangeListener()`, which is somewhat ill-documented. Assuming both the flag and `callChangeListener()` are true, the `Preference` should save its value to the persistent store via `persistInt()`, `persistString()`, or kin.

Using the Preference

Given all of that, using the custom `Preference` class in an application is almost anti-climactic. You simply add it to your preference XML, with a fully-qualified class name:

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <com.commonsware.cwac.colormixer.ColorPreference
    android:key="favoriteColor"
    android:defaultValue="0xFFA4C639"
    android:title="Your Favorite Color"
    android:summary="Blue.  No yel--  Auuuuuuuugh!" />
</PreferenceScreen>
```

At this point, it behaves no differently than does any other `Preference` type. Since `ColorPreference` stores the value as an integer, your code would use `getInt()` on the `SharedPreferences` to retrieve the value when needed.

The user sees an ordinary preference entry in the `PreferenceActivity`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

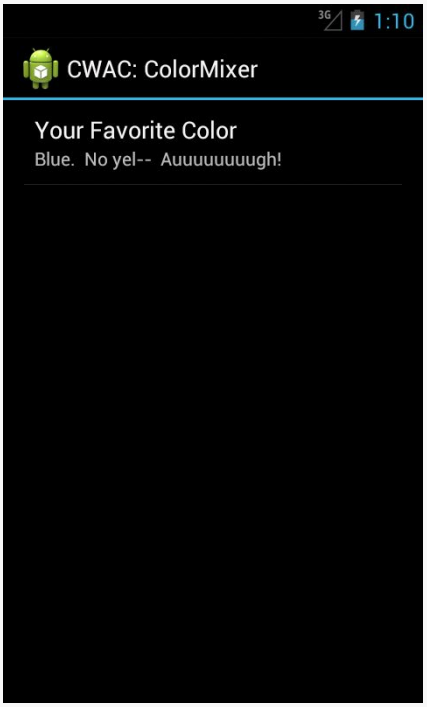


Figure 490: A PreferenceActivity, showing the ColorPreference

When tapped, it brings up the mixer:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

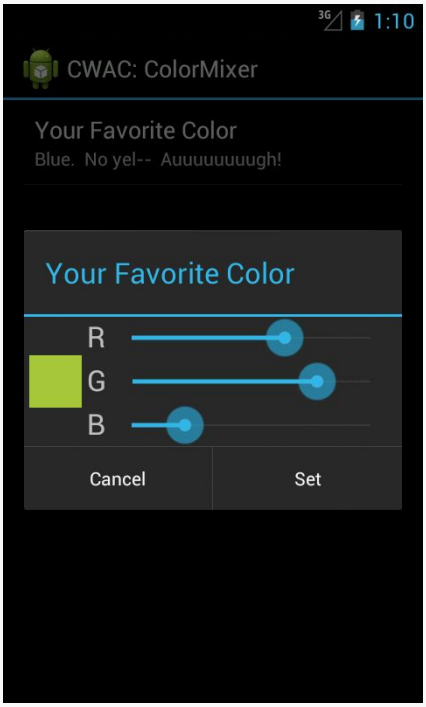


Figure 491: The ColorMixer in a custom DialogPreference

Choosing a color and clicking "Set" persists the color value as a preference.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Progress Indicators

Sometimes, we make the user wait. And wait. And wait some more.

Often, in these cases, it is useful to let the user know that something they requested is something that we are diligently working on. To do this, we can use some form of progress indicator. We saw basic use of a `ProgressBar` in the tutorials earlier in this book — now is the time to take a much closer look at `ProgressBar` and other means of displaying progress.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Having read the chapters on dialogs, custom drawables, and animators is also a good idea.

Progress Bars

The classic way to tell the user that we are doing something for them is to use a `ProgressBar` widget, much as we briefly displayed one in the EmPubLite sample app in the tutorials.

However, a `ProgressBar` is much more than a simple spinning image. We can use it to display either indeterminate progress ("we will be done... sometime") or specific progress ("we are 34% complete"). We can use it either as a circle or as a classic horizontal bar, the latter typically used for specific progress. And, for specific progress, we can actually show two tiers of progress, known as "primary" and "secondary" (e.g., primary for the progress in copying a directory's worth of files, secondary for the progress on a specific file).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In this section, we will take a look at these different ways of using `ProgressBar`.

Circular vs. Horizontal

As the name suggests, a `ProgressBar` denotes progress. As the name does not suggest, a `ProgressBar` is not a bar, by default — it is a circle. Hence, the following element from an XML layout resource:

```
<ProgressBar
  android:id="@+id/progressCI"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_gravity="center_horizontal"
  android:layout_marginBottom="20dp"
  android:layout_marginTop="20dp"/>
```

gives us:

Figure 492: Android 4.0 ProgressBar, Default Style

However, referencing `style="?android:attr/progressBarStyleHorizontal"` in the element:

```
<ProgressBar
  android:id="@+id/progressHI"
  style="?android:attr/progressBarStyleHorizontal"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_marginBottom="20dp"
  android:indeterminate="true"/>
```

gives us a horizontal bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 493: Android 4.0 ProgressBar, Horizontal Style

Note that the look-and-feel of these widgets have changed over the years. On Android 1.x and 2.x, they will look like this:

Figure 494: Android 2.3.3 ProgressBar, Both Styles

Specific vs. Indeterminate

Typically, you use the circular `ProgressBar` style for indeterminate progress, where the circle simply spins in place to let the user know that work is proceeding and the device (or activity) has not frozen. The horizontal `ProgressBar` style is used to illustrate specific amounts of progress, from 0 to a value you choose.

However, while those patterns are typical, the choice of whether to use indeterminate or some specific amount of progress is independent of the style of the widget.

The `android:indeterminate` attribute controls whether the `ProgressBar` will render an indeterminate look or a specific look. For the latter, calls to `setMax()` (or the `android:max` attribute) will set the upper end of the progress range (the default is 100), and `setProgress()` or `incrementProgressBy()` will set how much progress along that range is illustrated.

Figure 495: Android 4.0 ProgressBar, Horizontal Style, Indeterminate and Specific

Figure 496: Android 2.3.3 ProgressBar, Horizontal Style, Indeterminate and Specific

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Primary vs. Secondary

For specific progress, you actually have two independent amounts of progress. `setProgress()`, `incrementProgressBy()`, and `android:progress` control the primary progress, while `setSecondaryProgress()`, `incrementSecondaryProgressBy()`, and `android:secondaryProgress` control the secondary progress. Here, "primary progress" refers to the progress along an entire piece of work (e.g., copying a folder's worth of files), while "secondary progress" refers the progress along a discrete chunk of the overall work (e.g., copying an individual file).

A `ProgressBar` will render these with different colors, though primary trumps secondary, and so the secondary progress will only be visible when its value exceeds that of the primary progress:

Figure 497: Android 4.0 ProgressBar, Horizontal Style, Primary-Only and Primary-Plus-Secondary

Figure 498: Android 2.3.3 ProgressBar, Horizontal Style, Primary-Only and Primary-Plus-Secondary

ProgressBar and Threads

Normally, you cannot update the UI of a widget from a background thread.

`ProgressBar` is an exception. You *can* safely call `setProgress()` and `incrementProgressBy()` from a background thread to update the primary progress, and you can safely call `setSecondaryProgress()` and `incrementSecondaryProgressBy()` from a background thread to update the secondary progress.

To see this in action, take a look at the `Progress/BarSampler` sample project.

This project has a single activity (`MainActivity`), whose layout (`activity_main.xml`) contains four `ProgressBar` widgets, two indeterminate and two for specific progress:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <ProgressBar
    android:id="@+id/progressCI"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginBottom="20dp"
    android:layout_marginTop="20dp"/>

  <ProgressBar
    android:id="@+id/progressHI"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="20dp"
    android:indeterminate="true"/>

  <ProgressBar
    android:id="@+id/progressHS"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="20dp"
    android:indeterminate="false"
    android:max="100"/>

  <ProgressBar
    android:id="@+id/progressHS2"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:max="100"/>

</LinearLayout>
```

The activity gets access to the latter two `ProgressBar` widgets and sets up a `ScheduledThreadPoolExecutor` to get control every second in a background thread, which calls our `run()` method. The `run()` method will increment both `ProgressBar` widgets primary progress by 2 each time, and the secondary progress by 10 (dropping back to the starting point when the secondary progress reaches the maximum of 100). When the primary progress gets to 100, we cancel our scheduled work in the `ScheduledThreadPoolExecutor`:

```
package com.commonsware.android.progress;

import android.app.Activity;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.os.Bundle;
import android.widget.ProgressBar;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class MainActivity extends Activity implements Runnable {
  private static final int PERIOD_SECONDS=1;
  private ScheduledThreadPoolExecutor executor=
      new ScheduledThreadPoolExecutor(1);
  private ProgressBar primary=null;
  private ProgressBar secondary=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    primary=(ProgressBar)findViewById(R.id.progressHS);
    secondary=(ProgressBar)findViewById(R.id.progressHS2);

    executor.setExecuteExistingDelayedTasksAfterShutdownPolicy(false);
    executor.scheduleAtFixedRate(this, 0, PERIOD_SECONDS,
                                 TimeUnit.SECONDS);
  }

  @Override
  public void onDestroy() {
    executor.shutdown();

    super.onDestroy();
  }

  @Override
  public void run() {
    if (primary.getProgress() < 100) {
      primary.incrementProgressBy(2);
      secondary.incrementProgressBy(2);

      if (secondary.getSecondaryProgress() == 100) {
        secondary.setSecondaryProgress(10);
      }
      else {
        secondary.incrementSecondaryProgressBy(10);
      }
    }
    else {
      executor.remove(this);
    }
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The net effect is that you see the progress march across the screen, with the secondary progress going through five passes for the primary progress' single pass through the 0–100 range.

Tailoring Progress Bars

The stock `ProgressBar` look and feel is decent, if perhaps not spectacular. Often times, the stock look is sufficient for your needs. If you wish to have greater control over the look of your `ProgressBar`, the following sections will demonstrate some possibilities.

Changing the Progress Colors

The `ProgressBar` uses different colors for primary and secondary specific progress. By default, those colors are defined by the theme you are using, and the stock themes have firmware-defined colors (e.g., yellows for Android 1.x and 2.x, blues for Android 3.x and higher).

However, you can change the colors by using a `LayerListDrawable` and associating it with a `ProgressBar` by means of the `android:progressDrawable` attribute.

The `ProgressBar` background image needs to be a `LayerListDrawable` with three specific layers:

- `android:id="@android:id/background"` for the background color of the bar
- `android:id="@android:id/progress"` for the primary progress
- `android:id="@android:id/secondaryProgress"` for the secondary progress

Whether those layers are defined as `ShapeDrawable` structures, or as nine-patch PNG files is up to you, but they will need the ability to stretch to fit however big your bar winds up being.

To see what this means, let's take a look at the `Progress/Styled` sample project. This is a near-clone of the `Progress/BarSampler` project from earlier, using custom backgrounds for the bars. Here, we will look at the horizontal `ProgressBar` widgets — in the next section, we will look at how to change the background of a circular indefinite `ProgressBar`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For the first horizontal `ProgressBar` (`progressHS`), we will use a custom style created by Jérôme Van Der Linden's Android Holo Colors Generator, a Web site set up to help us create custom versions of the holographic widget theme.

When you visit this site in Google Chrome (note: other browsers are not supported at this time), you can fill in a name for your theme (e.g., "AppTheme"), the color scheme to use for the theme, and the foundation theme to use (light or dark):

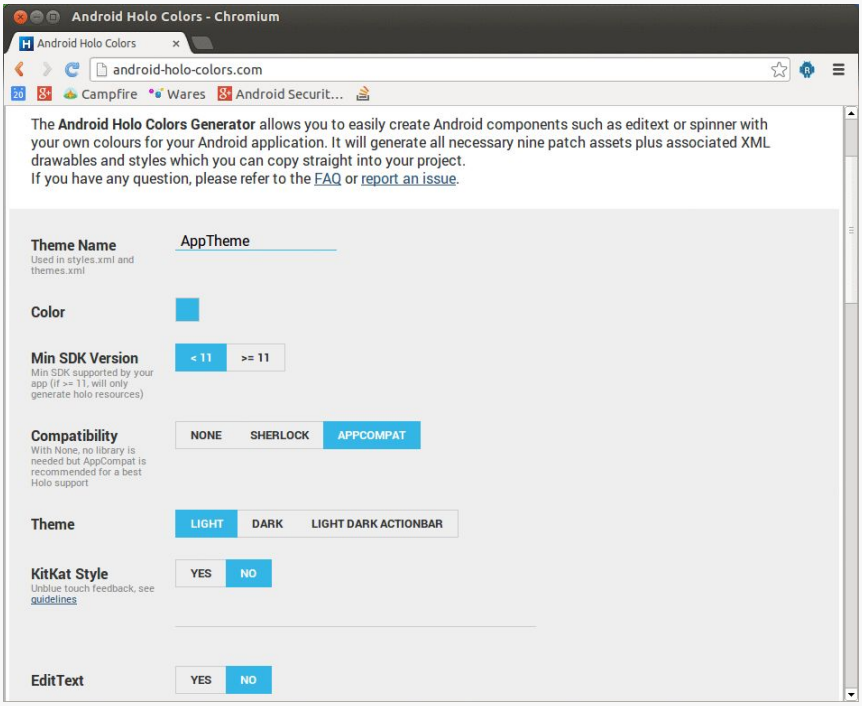


Figure 499: Android Holo Colors Generator, Basic Info

You can then toggle on and off which widgets you intend to use, so the generator will create custom styles for them:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

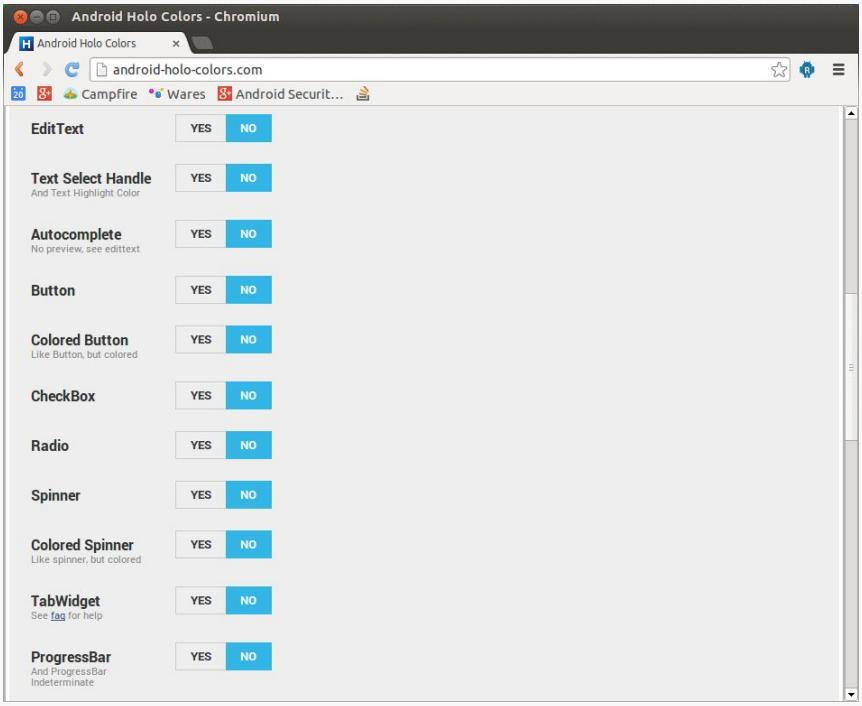


Figure 500: Android Holo Colors Generator, Widget Selection

Then, the generator will create a ZIP file that you can download that contains the generated resources for your custom styles.

The `Progress/Styled` project contains the files generated by the generator, replacing the original style resources. Note that the generator does not create a `.DarkActionBar` version of the style resource, so the `values-v14` resource directory in the project has one hand-crafted based upon a regular generated style resource.

Our manifest points to our `AppTheme` as being how we wish to style widgets in this application:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.progress"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="15"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:theme="@style/AppTheme">
    <activity
      android:name=".MainActivity"
      android:label="@string/title_activity_main">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

That theme, defined in `apptheme_themes.xml`, points to style resources for horizontal `ProgressBar` widgets:

```
<?xml version="1.0" encoding="utf-8"?>

<!-- Generated (in part) with http://android-holo-colors.com -->
<resources xmlns:android="http://schemas.android.com/apk/res/android">

  <style name="AppTheme" parent="android:Theme.Holo.Light.DarkActionBar">

    <item name="android:progressBarStyleHorizontal">@style/
ProgressBarAppTheme</item>

  </style>

</resources>
```

The `ProgressBarAppTheme` style resource is defined in a separate `apptheme_styles.xml` resource:

```
<?xml version="1.0" encoding="utf-8"?>

<!-- Generated with http://android-holo-colors.com -->
<resources xmlns:android="http://schemas.android.com/apk/res/android">

  <style name="ProgressBarAppTheme"
parent="android:Widget.Holo.Light.ProgressBar.Horizontal">
      <item name="android:progressDrawable">@drawable/
progress_horizontal_holo_light</item>
      <item name="android:indeterminateDrawable">@drawable/
progress_indeterminate_horizontal_holo_light</item>
  </style>

</resources>
```

Here, we say that we want the `android:progressDrawable` property to be a `progress_horizontal_holo_light` drawable resource. We also set the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`android:indeterminateDrawable` property — used for indeterminate bars — to a `progress_indeterminate_horizontal_holo_light` drawable resource.

Those are defined as XML-based drawables, in the `res/drawable/` directory in the project. The `progress_horizontal_holo_light` resource is defined as:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2010 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->

<layer-list xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:id="@android:id/background"
          android:drawable="@drawable/progress_bg_holo_light" />

    <item android:id="@android:id/secondaryProgress">
        <scale android:scaleWidth="100%"
               android:drawable="@drawable/progress_secondary_holo" />
    </item>

    <item android:id="@android:id/progress">
        <scale android:scaleWidth="100%"
               android:drawable="@drawable/progress_primary_holo" />
    </item>

</layer-list>
```

The generator creates our `LayerListDrawable` resource with our three layers, each pointing to a nine-patch PNG file (with different versions for different densities) that contains our desired custom color. The `progress` and `secondaryProgress` layers use `ScaleDrawable` definitions to ensure that the images are measured against the complete width of the background layer, which in turn will be sized according to the size of the `ProgressBar` itself.

We will take a look at the `progress_indeterminate_horizontal_holo_light` drawable resource in the next section.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that you could skip the custom theme and style if you wished, and simply add the `android:progressDrawable` attribute to the `ProgressBar` widget definition in its layout XML resource.

Regardless, the result is that our progress bars have the desired purple color scheme:

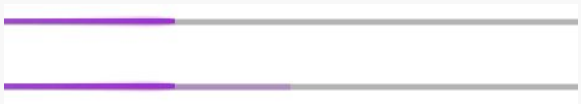


Figure 501: Custom ProgressBar Style, Primary and Secondary

Also, you can have your `LayerListDrawable` use `ShapeDrawable` layers, to avoid creating nine-patch PNG files, if you prefer, using a resource like this:

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@android:id/background">
        <shape>
            <stroke android:width="1dip" android:color="#FF333333" />
            <gradient
                    android:startColor="#FF9C9E9C"
                    android:centerColor="#FF5A5D5A"
                    android:centerY="0.71"
                    android:endColor="#FF6B716B"
                    android:angle="270"
            />
        </shape>
    </item>
    <item android:id="@android:id/secondaryProgress">
        <clip>
            <shape>
            <stroke android:width="1dip" android:color="#FF333333" />
                <gradient
                        android:startColor="#4cffffff"
                        android:centerColor="#4cE7E7E7"
                        android:centerY="0.71"
                        android:endColor="#4cFFFBFF"
                        android:angle="270"
                />
            </shape>
        </clip>
    </item>
    <item android:id="@android:id/progress">
        <clip>
            <shape>
                <stroke android:width="1dip" android:color="#FF333333" />
                <gradient
                        android:startColor="#FFFFFFFF"
                        android:centerColor="#FFE7E7E7"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                        android:centerY="0.71"
                        android:endColor="#FFFFFBFF"
                        android:angle="270"
                />
            </shape>
        </clip>
    </item>
</layer-list>
```

Changing the Indeterminate Animation

Similarly, for indefinite progress "bars", changing the progress drawable will let you change the way they look. However, in this case, the drawable also needs to implement the animation itself. You can accomplish this either by using an `AnimationDrawable` or by using some other type of drawable wrapped in an animation, such as a `ShapeDrawable` wrapped in a `<rotate>` animation.

For example, the custom theme created by the Android Holo Colors Generator assigns the following drawable resource to `android:indeterminateDrawable` in the theme:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
/*
** Copyright 2011, The Android Open Source Project
**
** Licensed under the Apache License, Version 2.0 (the "License");
** you may not use this file except in compliance with the License.
** You may obtain a copy of the License at
**
**     http://www.apache.org/licenses/LICENSE-2.0
**
** Unless required by applicable law or agreed to in writing, software
** distributed under the License is distributed on an "AS IS" BASIS,
** WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
** See the License for the specific language governing permissions and
** limitations under the License.
*/
-->
<animation-list
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:oneshot="false">
    <item android:drawable="@drawable/progressbar_indeterminate_holo1"
android:duration="50" />
    <item android:drawable="@drawable/progressbar_indeterminate_holo2"
android:duration="50" />
    <item android:drawable="@drawable/progressbar_indeterminate_holo3"
android:duration="50" />
    <item android:drawable="@drawable/progressbar_indeterminate_holo4"
android:duration="50" />
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    <item android:drawable="@drawable/progressbar_indeterminate_holo5"
android:duration="50" />
    <item android:drawable="@drawable/progressbar_indeterminate_holo6"
android:duration="50" />
    <item android:drawable="@drawable/progressbar_indeterminate_holo7"
android:duration="50" />
    <item android:drawable="@drawable/progressbar_indeterminate_holo8"
android:duration="50" />
</animation-list>
```

Hence, every horizontal indeterminate `ProgressBar` will use that `AnimationDrawable`. The individual images in the animation are PNG files, with different versions for different densities.

Circular `ProgressBar` widgets also need a custom progress drawable, though obviously the image will need to be circular, not a bar. You can certainly use an `AnimationDrawable` for this, or you can use a `ShapeDrawable`, such as the `res/drawable/progress_circular.xml` resource shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<rotate xmlns:android="http://schemas.android.com/apk/res/android"
  android:fromDegrees="0"
  android:pivotX="50%"
  android:pivotY="50%"
  android:toDegrees="360">

  <shape
    android:innerRadiusRatio="3"
    android:shape="ring"
    android:thicknessRatio="8"
    android:useLevel="false">
    <gradient
      android:centerColor="#4c737373"
      android:centerY="0.50"
      android:endColor="#ff9933CC"
      android:startColor="#4c737373"
      android:type="sweep"
      android:useLevel="false"/>
  </shape>

</rotate>
```

Here, we have a `ring` `ShapeDrawable`, with a certain thickness and radius, filled with a gradient. Half of the fill is actually a solid color (`#4c737373`), as the start and center colors are the same. The other half is a sweep gradient from the starting color to the same purple shade that is used by the other bar styles. This ring is then wrapped in a `rotate` animation. This yields a simple gradient-filled ring, that rotates smoothly to indicate progress:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

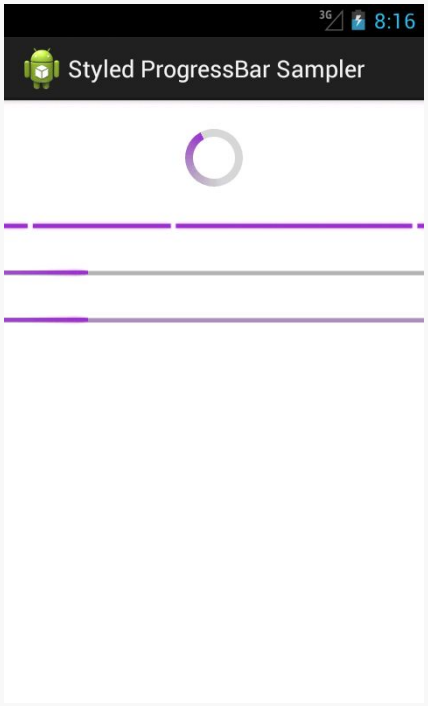


Figure 502: Custom ProgressBar Styles, Including Circular Indefinite

Note that the Android Holo Colors Generator does not generate circular indefinite `ProgressBar` resources as of the time of this writing.

Progress Dialogs

One use of a `ProgressBar` is to have it wrapped in a `ProgressDialog`. Like all dialogs, `ProgressDialog` is modal, preventing the user from interacting with an underlying activity while the dialog is displayed. From a UI design standpoint, a `ProgressDialog` is an easy way to temporarily show progress without having to find a spot for a `ProgressBar` widget somewhere in the UI. Also, since usually there are things in the activity that are dependent upon the work being done in the background, having the dialog in place prevents anyone from trying to use things that are not yet ready.

However, modal dialogs are not a great design approach, as they aggressively limit the user's options. `ProgressDialog` is perhaps the worst in this regard, as the user can do nothing except wait. While *part* of your app may not yet be ready, other parts surely are, such as reading the documentation, or adjusting settings, or clicking on

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

your ad banners. Hence, using *anything* else other than `ProgressDialog`, while perhaps a bit more work, will be an improvement in the usability of your app.

That being said, let us see how to set up a `ProgressDialog`. The `Progress/Dialog` sample project is a near-clone of the `Dialogs/DialogFragment` sample project from the chapter on dialogs. The only difference is the `onCreateDialog()` method of our `DialogFragment`, where we directly create a `ProgressDialog` instead of using an `AlertDialog.Builder` to create an `AlertDialog` as before:

```
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
  ProgressDialog dlg=new ProgressDialog(getActivity());

  dlg.setMessage(getActivity().getString(R.string.dlg_title));
  dlg.setIndeterminate(true);
  dlg.setProgressStyle(ProgressDialog.STYLE_SPINNER);

  return(dlg);
}
```

We create the `ProgressDialog` via its constructor, set the message explaining what we are waiting for via `setMessage()`, indicate that the `ProgressBar` should be an indeterminate one via `setIndeterminate()`, and indicate that we want a circular "spinner" `ProgressBar` rather than a horizontal one by calling `setProgressStyle(ProgressDialog.STYLE_SPINNER)`. There are a variety of other things you could configure on the `ProgressDialog` if desired, and `ProgressDialog` inherits from `AlertDialog`, so some things you could configure on an `AlertDialog` will also be available on the `ProgressDialog`.

The result is a dialog that you may have seen from other apps in Android:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

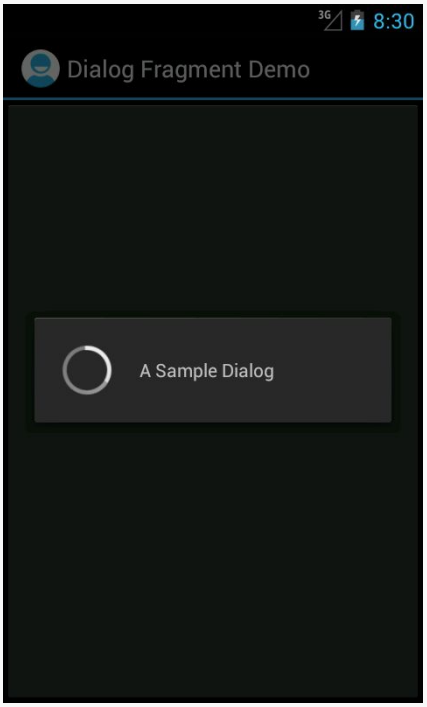


Figure 503: ProgressDialog

Title Bar and Action Bar Progress Indicators

Another place to let users know that you are doing something on their behalf is to put a progress indicator in the title bar or action bar of your activity. This avoids your having to put an indeterminate `ProgressBar` somewhere in your activity's UI. It is also very simple to set up, as we can see in the Progress/TitleBar sample project.

```
package com.commonsware.android.titleprog;

import android.app.Activity;
import android.os.Bundle;
import android.view.Window;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    getWindow().requestFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_main);
    setProgressBarIndeterminateVisibility(true);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Up front, as the first thing that you do in your `onCreate()` call, you need to call:

```
getWindow().requestFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
```

This tells Android to reserve space in your title bar or action bar for an indeterminate progress indicator, though the indicator does not appear at this point.

Later on, when you want the indicator to actually appear, call `setProgressBarIndeterminateVisibility(true)` on your activity, and later call `setProgressBarIndeterminateVisibility(false)` to make the indicator go away.

This particular application has `android:targetSdkVersion` set to `11` or higher, but it is not using an action bar backport like ActionBarSherlock. Hence, when you run it on an older Android environment, you get a classic title bar with the progress indicator on the right:

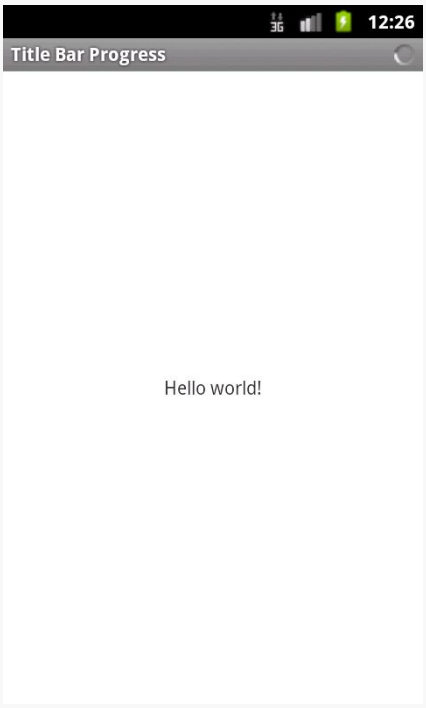


Figure 504: Progress Indicator in Title Bar

When you have an action bar, you get the same basic effect, albeit with a larger indicator to match the larger bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

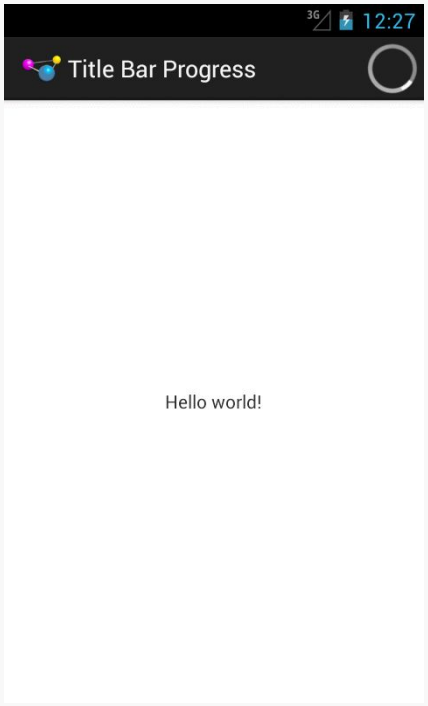


Figure 505: Progress Indicator in Action Bar

Direct Progress Indication

Sometimes, the best way to let the user know about updates is to simply update the data in place. Rather than have some separate indicator, let the core UI itself convey the work being done.

We saw this in the chapter on threads, where we populated a `ListView` in "real time" as we loaded in data into its adapter. Other variations on this theme include:

- Updating a page count `TextView` to show the number of downloaded pages, while the user is reading earlier pages, perhaps with some sort of style (e.g., italics) or color coding (e.g., red) to indicate data that is being loaded.
- Simply disabling the buttons, action bar items, and other ways that the user could navigate to a point in your app where you need the data that is being loaded in the background. The key here is to make sure that users understand *why* those items are disabled, and sometimes that is not obvious. Hence, while this step may be necessary, it is often tied in with progress indicators in the title bar or action bar or other means of indicating to the user the *reason* they cannot perform certain operations.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Advanced Notifications

Notifications are those icons that appear in the status bar (or system bar on tablets), typically to alert the user of something that is going on in the background or has completed in the background. Many apps use them, to let the user know of new email messages, calendar reminders, and so on. Foreground services, such as music players, also use notifications, to tell the OS that they are part of the foreground user experience and to let the user rapidly return to the apps to turn the music off.

There are other tricks available with the `Notification` object beyond those originally discussed in an earlier chapter.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the chapter on basic notifications and the section on `RemoteViews` in the chapter on basic app widgets.

Being a Good Citizen

Users have a love/hate relationship with apps that use notifications:

- They love apps that raise notifications for events that the user cares about...
- ...but they hate apps that raise notifications for events that the user does not care about (e.g., Evernote's "please confirm your email" notifications)
- They love apps that provide control over when and how notifications appear...
- ...but they hate apps that display notifications solely because the developer wanted them (e.g., ads in notifications)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- They love apps that use notifications to let the user control some background operation, like media playback...
- ...but they hate apps that have ongoing notifications for no obvious reason (e.g., developers trying to use a foreground service to keep their process around, rather than using `AlarmManager`, `JobScheduler`, or other means of doing work periodically)
- They love apps that set up notifications for use in different scenarios, such as supporting Android Wear devices...
- ...but they hate apps that wind up flooding their wrist (or eyes, or other wearable locations) with notifications that have to be individually dismissed

And so on.

Users' discomfort with how apps handle notifications is why Android allows users to disable notifications.

Some of the items in this chapter, particularly those surrounding Android Wear, can help you improve user satisfaction with your notification strategy and tactics. Yet, at the same time, misuse of notifications is magnified by Wear, as Wear takes extra steps to get the user to pay attention to the notifications, with possibly disastrous results for your Play Store reviews.

In short, your objective with notifications is to be a good citizen:

- Have a reasonable default mode for your notifications
- Allow users to tailor that mode to better suit their needs, where practical

Wear? There!

The humble `Notification` has been steadily advancing over the past few years, with "big" styles and the like adding new capabilities for newer devices.

Android Wear takes notifications to a new level, by having the notification not only appear on the user's device, but also on wearables connected to that device.

The good news is that this works "out of the box". There is nothing you absolutely need to do in your app to get your notifications to appear on a Wear device.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The bad news is that the "out of the box" experience may be poor, as a `Notification` approach that is fine for devices that reside in pockets and backpacks might be inappropriate for wrists and eyes.

With that in mind, let's see what some notification samples from earlier in the book behave like when they are run on a phone connected to a Wear device.

NOTE: For this section, and the rest of this chapter, "primary device" will refer to the user's phone or tablet that the "Wear device" will be connected to.

Simple Notification

The `Notifications/DownloadNotify` sample project allows the user to download a PDF file, raising a `Notification` when that download is complete.

With a Wear device paired with the phone, the `Notification` also appears on the device, first as a "mini card":

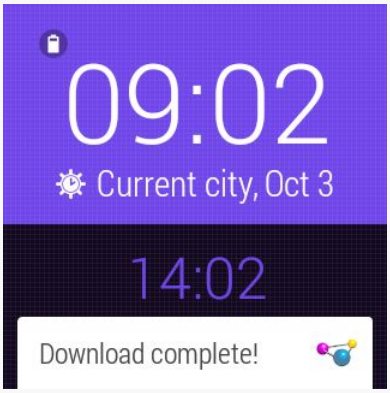


Figure 506: Simple Notification on Wear, As Originally Displayed, On Samsung Galaxy Gear

Swiping up on that will bring up the full card:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

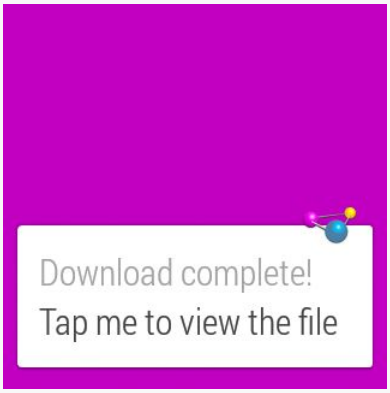


Figure 507: Simple Notification on Wear, Full, On Samsung Galaxy Gear

Swiping to the right will bring up the action associated with `setContentIntent()` on the `NotificationCompat.Builder`:

Figure 508: Simple Notification on Wear, Default Action, On Samsung Galaxy Gear

Tapping on that dismisses the `Notification` on the Wear device and the primary device, plus it invokes the `PendingIntent` on the phone itself (in this case, opening up the PDF file).

This is a fine example of a `Notification` that perhaps should *not* appear on the Wear device. The fact that the download completed is interesting but not all that important. Furthermore, the user cannot do anything about this download other than to pull out the primary device to see the PDF. Low-priority primary-device-centric notifications generally should be shown on the primary device alone, not on the Wear device. We will see how to do that later in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

"Big" Style and Action Button

The `Notifications/BigNotify` sample application wrapped a regular `Notification` in a `NotificationCompat.InboxStyle` "big" `Notification`, one with both a regular action and a separate "Play" action button.

As before, with a Wear device paired with the phone, the `Notification` also appears on the device, first as a "mini card":

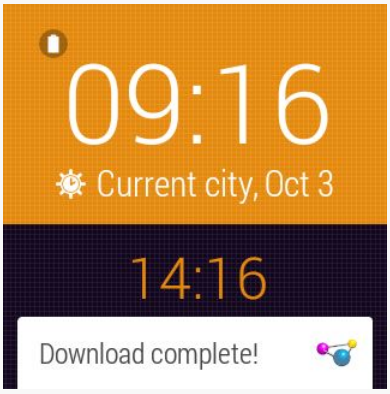


Figure 509: Big Notification on Wear, As Originally Displayed, On Samsung Galaxy Gear

However, this time, when the user swipes up to show the full card, it is the `InboxStyle` version that appears, albeit without the summary text:

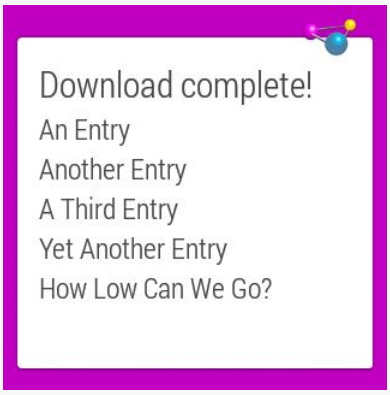


Figure 510: Big Notification on Wear, Full, On Samsung Galaxy Gear

Swiping to the right shows our actions, starting with the custom "Play" action:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

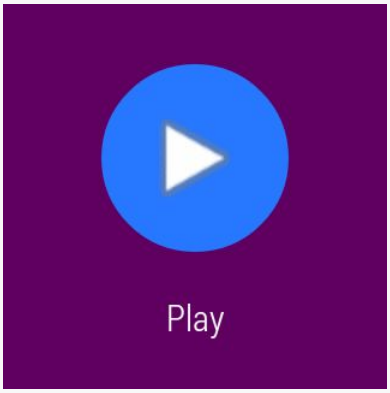


Figure 511: Big Notification on Wear, Play Action, On Samsung Galaxy Gear

...followed by the default action:

Figure 512: Big Notification on Wear, Default Action, On Samsung Galaxy Gear

Tapping on either action will cause the primary device to invoke its `PendingIntent`, but only the default action dismisses the `Notification` from both devices. The custom "Play" action does not.

Foreground Service

The `Notifications/Foreground` sample project is another version of the download-the-file sample, but this time uses a `Notification` and `startForeground()` to mark the service as a foreground service while it is downloading things.

This particular sample does not spend much time in the foreground state, so for testing purposes, you may want to add a `SystemClock.sleep()` call to the service,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

between the `startForeground()` and `stopForeground()` calls, to better examine the behavior while the foreground service `Notification` is around.

However, in truth, that modification is probably not necessary... as the foreground service `Notification` is not displayed on the Wear device, only on the primary device. This is by design. The expectation is that you would use a Wear app to control your service from the Wear device, not some un-dismissable card.

Stacking Notifications

If you are writing an email client, and you want to use a `Notification` to let the user know about new email messages, you do *not* want to raise a separate `Notification` for each email. Users will come to your home with pitchforks and torches... and not to help you with farming.

Instead, the vision is that you update an existing `Notification` with new content. For example, you might start with a regular `Notification` for the first received email. Then, when the second one comes in, you replace that `Notification` with one that has a simple summary ("2 messages are in your inbox!"), plus perhaps an `InboxStyle` "big" `Notification` variant that could show the subject lines for both of those messages.

Android Wear devices, however, add an interesting wrinkle: you want the `Notification` to be informative about the event itself. You want the user to be able to make an informed decision about whether they should pull out their primary device to read the new messages, and that decision is only partly based on how many messages there are. Users will want to know more about the outstanding messages (sender and/or subject line) to help them make that decision... at least to a point. If there are 57 unread messages, users may get frustrated dealing with all of those as individual items on the wearable itself.

The pattern here, then, takes advantage of some "group" capabilities added to `NotificationCompat`:

- Raise one "summary" `Notification`, that will only be shown on the primary device, with the same sort of "2 messages are in your inbox!" information that you would have used without considering Wear
- Raise individual notifications for individual messages that will appear on the Wear device

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Collect all of those in a "group", so the primary device shows only the summary and the Wear device shows only the individual ones

This can be seen in action in the `Notifications/Stacked` sample project.

The setup is reminiscent of the "big" style one from the original chapter on `Notification`. However, this time, there are a total of three `Notification` objects created: two for individual events for the Wear device, and one summary one for the primary device.

However, to make this work, we need a new version of the `support-v13` library from the Android Support package: `20.0.0` (or higher), as it is where the extra compatibility smarts were added to support this whole group-and-summary construct. Hence, in `build.gradle`, we have `compile 'com.android.support:support-v13:20.0.0'`, and for Eclipse, we have the corresponding JAR in `libs/`.

Similarly, while we will still use `NotificationCompat` for creating the `Notification` objects, we will *not* use `NotificationManager` for displaying them. Instead, we need to use `NotificationManagerCompat` from the Android Support package. While the `NotificationManager` API has not changed to support the group-and-summary pattern, the *implementation* has, and `NotificationManagerCompat` gives us a version of that implementation that can work on compatible devices and gracefully degrade on older ones. However, since the API did not change, it is easy to miss this requirement, use `NotificationManager`, and not quite get the desired results. Notably, the primary device will wind up showing all three notifications, not just the summary as we want.

Hence, our `MainActivity` will hold onto a `NotificationManagerCompat` as a data member, initialized in `onCreate()`:

```
private NotificationManagerCompat mgr=null;

@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  mgr=NotificationManagerCompat.from(this);

  showWearOne();
  showWearTwo();
  showSummary();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    finish();
  }
```

The three `show...()` methods are each responsible for raising one `Notification`: `showWearOne()` and `showWearTwo()` are ones that will wind up on the Wear device, and `showSummary()` will show the summary `Notification` for use on the primary device.

Beyond using `NotificationManagerCompat` instead of `NotificationManager`, the only substantial difference is the use of `setGroup()` and `setGroupSummary()` methods on the `NotificationCompat.Builder`.

`setGroup()` associates the `Notification` with a group, identified by a `String` key. On a Wear device, notifications that are part of a group will be shown stacked as part of a single card by default. So, the two `showWear...()` methods call `setGroup()` as part of building the `Notification`:

```
  private void showWearOne() {
    NotificationCompat.Builder b=new NotificationCompat.Builder(this);

    b.setAutoCancel(true)
        .setDefaults(Notification.DEFAULT_ALL)
        .setContentTitle(getString(R.string.entry))
        .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
        .setSmallIcon(android.R.drawable.stat_sys_download_done)
        .setTicker(getString(R.string.download_complete))
        .setGroup(GROUP_SAMPLE);

    mgr.notify(NOTIFY_ID2, b.build());
  }

  private void showWearTwo() {
    NotificationCompat.Builder b=new NotificationCompat.Builder(this);

    b.setAutoCancel(true)
        .setDefaults(Notification.DEFAULT_ALL)
        .setContentTitle(getString(R.string.another_entry))
        .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
        .setSmallIcon(android.R.drawable.stat_sys_download_done)
        .setTicker(getString(R.string.download_complete))
        .setGroup(GROUP_SAMPLE);

    mgr.notify(NOTIFY_ID3, b.build());
  }

  private PendingIntent buildPendingIntent(String action) {
    Intent i=new Intent(action);

    return(PendingIntent.getActivity(this, 0, i, 0));
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`setGroupSummary()` indicates a particular `Notification` that should serve as the summary for its group. This `Notification` will not be passed to the Wear device, and it replaces all other notifications for this group on the primary device. Hence, `showSummary()` (or, more accurately, the `buildNormal()` method that creates the base `Notification` for the summary) uses `setGroupSummary()`:

```
  private void showSummary() {
    NotificationCompat.Builder normal=buildNormal();
    NotificationCompat.InboxStyle big=
        new NotificationCompat.InboxStyle();

    big.setSummaryText(getString(R.string.summary))
        .addLine(getString(R.string.entry))
        .addLine(getString(R.string.another_entry));

    mgr.notify(NOTIFY_ID, normal.setStyle(big).build());
  }

  private NotificationCompat.Builder buildNormal() {
    NotificationCompat.Builder b=new NotificationCompat.Builder(this);

    b.setAutoCancel(true)
        .setDefaults(Notification.DEFAULT_ALL)
        .setContentTitle(getString(R.string.download_complete))
        .setContentText(getString(R.string.fun))
        .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
        .setSmallIcon(android.R.drawable.stat_sys_download_done)
        .setTicker(getString(R.string.download_complete))
        .setGroup(GROUP_SAMPLE)
        .setGroupSummary(true);

    return(b);
  }
```

Note that you need to use `setGroupSummary()` on a `NotificationCompat.Builder` on which you have also called `setGroup()`, to identify the group for which this `Notification` is a summary.

When you run this, the primary device shows the summary `Notification`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

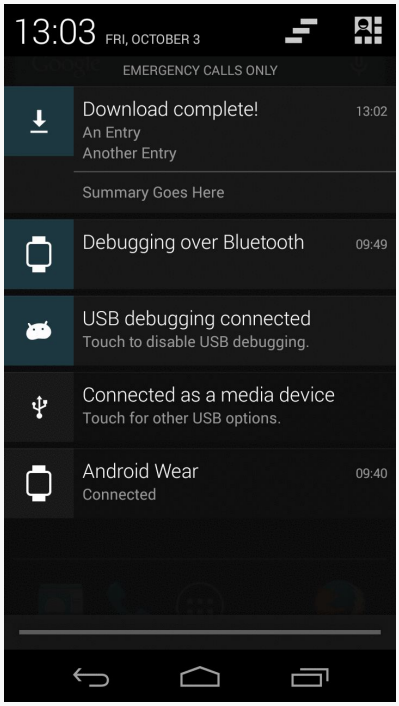


Figure 513: Stacked Notifications, Summary on Primary Device

On the Wear device, you will see the two original notifications as part of a single card at the outset:

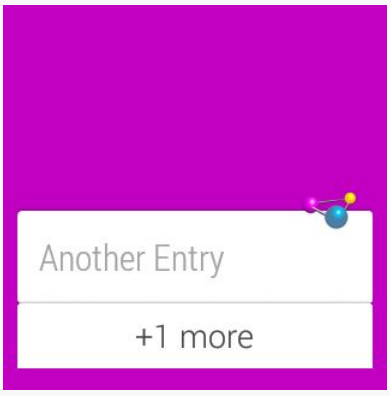


Figure 514: Stacked Notifications, Stacked on Wear Device

Tapping on the stack brings up separate mini cards for each individual `Notification`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

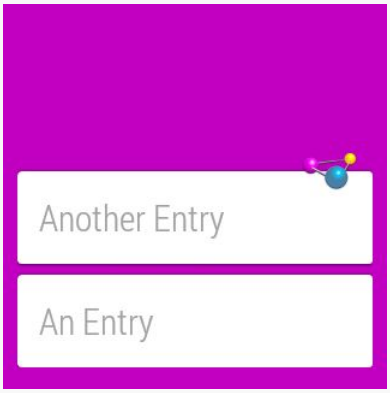


Figure 515: Stacked Notifications, Expanded Stack on Wear Device

…And the Passage of Time

Of course, this sample is artificially simple, like most of the samples in this book.

In the sample, we are raising all three notifications all at once. That is certainly conceivable, but it is not especially likely. A more likely scenario is that the mix of notifications needs to change over time, based upon continuing events, such as a trickle of new unread email messages for an email client.

This adds a few complexities to what you need to implement all of this properly.

The big thing is that your persistent data model (e.g., database) needs to have enough information for you to know how to notify the user about the next event, when that event occurs. Using the email client as an example:

- We start off in the "steady state" of no unread email messages and, therefore, no notifications from our app.
- A new email message arrives. At this point, we want to show a regular `Notification` on both the Wear device and the primary device, with the sender and subject line of the unread message.
- A second new email message arrives later. At this point, we want to show another regular `Notification` (requiring a separate notification ID) for the Wear device, but also show a summary `Notification` for the primary device. For all that to work, we need to know this *is* a second unread message, and that the user has not read the first message in between the two incoming messages. And, we need to know enough details about the unread messages to format the summary properly.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This gets even more complex when events "stack themselves" (e.g., one poll of the mail server results in two unread messages), in addition to having to deal with user input (e.g., user clears the notification stack from either device, yet does not read the messages).

Among other things, you cannot rely upon static data members as being the sole source of your `Notification`-related data, as your process may be terminated in between events. You are welcome to use it as a cache, in case your process does happen to survive long enough to process more than one event, but you will need to also save this data to a persistent store, so that you can properly handle new events requiring `Notification` changes with your process having been terminated since the last `Notification`-related event.

Avoiding Wear

Sometimes, you will want to raise a `Notification` that does not make sense to show on a Wear device, only on the primary device. In the case of the group summary for the stacked notifications, this primary-only behavior happens automatically. In other cases, though, you will need to call `setLocalOnly()` on the `NotificationCompat.Builder` to tell the framework that this `Notification` should only be displayed on the current device.

The `Notifications/BigLocal` sample project demonstrates this, through a clone of the `Notifications/BigNotify` sample that has just two changes:

1. It switches to the `20.0.0` version of the `support-v13` library, to get a version of `NotificationCompat.Builder` that offers `setLocalOnly()`
2. It calls `setLocalOnly(true)` as part of configuring the `Notification`:

```
  private NotificationCompat.Builder buildNormal() {
    NotificationCompat.Builder b=new NotificationCompat.Builder(this);

    b.setAutoCancel(true)
     .setDefaults(Notification.DEFAULT_ALL)
     .setContentTitle(getString(R.string.download_complete))
     .setContentText(getString(R.string.fun))
     .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
     .setSmallIcon(android.R.drawable.stat_sys_download_done)
     .setTicker(getString(R.string.download_complete))
     .setPriority(Notification.PRIORITY_HIGH)
     .setLocalOnly(true)
     .addAction(android.R.drawable.ic_media_play,
         getString(R.string.play),
         buildPendingIntent(Settings.ACTION_SETTINGS));
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(b);
  }
```

Note that we do not need to use `NotificationManagerCompat` for local-only behavior — simply calling `setLocalOnly(true)` on an up-to-date `NotificationCompat.Builder` will suffice.

Running this sample provides the same behavior as `Notifications/BigNotify`, except that the `Notification` only appears on the primary device, not the Wear device.

Other Wear-Specific Notification Options

Configuring stacked notifications, and opting into local-only behavior when needed, should give you Wear behavior that is acceptable. Right now, Android Wear is fairly nascent, and therefore it may not behoove you to do much more than this, as you decide how to prioritize your engineering time.

However, there are other things that you can do to further tailor your notifications on Wear that can improve user satisfaction, if you wish for Wear to be a key part of your marketing message.

Pages

On the primary device, the amount of information you can provide in a `Notification` is intentionally capped. This prevents a `Notification` from drowning out its peers. The cap is not a big problem, simply because the whole UI for the app raising the `Notification` is usually just a tap away.

With a Wear device, though, the whole UI for the app raising the `Notification` involves pulling out the primary device.

Hence, it might be nice to provide some additional information to the Wear user, so that perhaps they can make a more informed decision as to whether it is worthwhile to open up their primary device. In Wear terms, this involves adding more “pages” to a `Notification`.

To do this, you must:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Create the second (and additional) pages as their own separate `Notification` objects, probably via a `NotificationCompat.Builder`
- Use a `NotificationCompat.WearableExtender` to teach the primary `Notification` about the additional pages
- Raise the primary `Notification` using a `NotificationManagerCompat` variant of the system service

We can see this in action in the [Notifications/Pages] sample project. This is a clone of `Notifications/BigNotify`, where we make the "big" content be on a second page.

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  NotificationManagerCompat mgr=
      NotificationManagerCompat.from(this);
  NotificationCompat.Builder normal=buildNormal();
  NotificationCompat.InboxStyle big=
      new NotificationCompat.InboxStyle();

  big.setSummaryText(getString(R.string.summary))
      .addLine(getString(R.string.entry))
      .addLine(getString(R.string.another_entry))
      .addLine(getString(R.string.third_entry))
      .addLine(getString(R.string.yet_another_entry))
      .addLine(getString(R.string.low));

  NotificationCompat.Builder bigPage=
      new NotificationCompat.Builder(this)
          .setStyle(big);
  NotificationCompat.Builder twoPages=
      new NotificationCompat.WearableExtender()
          .addPage(bigPage.build())
          .extend(normal);

  mgr.notify(NOTIFY_ID, twoPages.build());

  finish();
}
```

Here, we:

- Create a `NotificationManagerCompat` instance
- Create the primary ("normal") `Notification`, using the same process as before
- Create the `InboxStyle` structure with our expanded content

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Wrap that "big" style in another `Notification` via a `NotificationCompat.Builder`, using the `setStyle()` method to associate the "big" style with the `Notification`
- Create a `NotificationCompat.WearableExtender`, tell it to add the second page using `addPage()`, and tell it to apply that second page to the primary `Notification` via the `extend()` method
- Use `notify()` as normal to raise the `Notification`, using the already-created `NotificationManagerCompat` instance

On the primary device, we just see the primary `Notification` content:

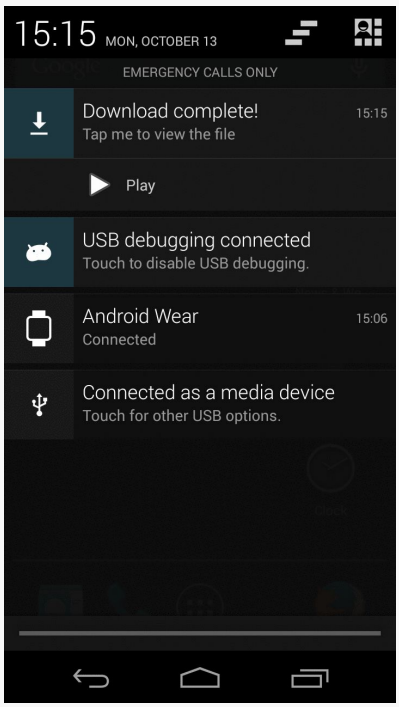


Figure 516: Pages Demo, on a Galaxy Nexus

On the Wear device, we see the main `Notification` and the second page as separate pages on the wearable:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

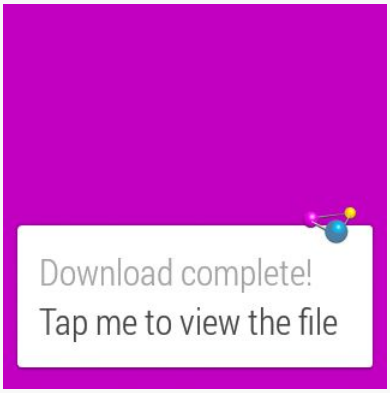


Figure 517: Pages Demo, on a Samsung Galaxy Wear, Showing Initial Notification

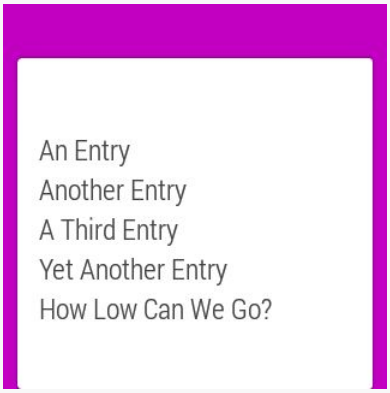


Figure 518: Pages Demo, on a Samsung Galaxy Wear, Showing Second Page

Note that you cannot use `addAction()` to define a custom action on the extra pages added to the primary `Notification`. Instead, use `addAction()` and `setContentAction()` on the `WearableExtender` to define actions associated with those extra pages. We will see this in use in the next section.

Wear-Only Actions

Sometimes, you may want certain actions to only be available on the Wear device, and not on the primary device. We will see a specific example of this coming up in the next section, when we cover voice input actions.

Sometimes, you may want a different mix of actions on the primary device versus the Wear device — some in common, some only on the primary device, some only on the Wear device.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To set up Wear-only actions, use `addAction()` on `WearableExtender`, as opposed to (or in addition to) `addAction()` on `NotificationCompat.Builder`. This takes an action as a parameter, which you create using `NotificationCompat.Action.Builder`, a custom builder for building `Notification` actions.

This is illustrated in the `Notifications/WearActions` sample project, yet another variation on the "launch an activity, show a `Notification`" samples that we have been using. This time, though, we will apply an action to the Wear device:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    NotificationManagerCompat mgr=
        NotificationManagerCompat.from(this);
    NotificationCompat.Builder normal=buildNormal();
    NotificationCompat.Action.Builder wearActionBuilder=
        new NotificationCompat.Action.Builder(android.R.drawable.ic_media_pause,
                                              getString(R.string.pause),

buildPendingIntent(Settings.ACTION_DATE_SETTINGS));

    NotificationCompat.Builder extended=
        new NotificationCompat.WearableExtender()
            .addAction(wearActionBuilder.build())
            .extend(normal);

    mgr.notify(NOTIFY_ID, extended.build());

    finish();
  }
```

Here, we:

- Create a `NotificationManagerCompat` instance
- Create the primary ("normal") `Notification`, using the same process as before
- Create an instance of `NotificationCompat.Action.Builder`, providing it the icon, label, and `PendingIntent` to be invoked for this action
- Create an instance of `NotificationCompat.WearableExtender`, adding the newly-defined action to it, and using the `WearableExtender` to `extend()` the primary `Notification`
- Show that extended `Notification` using the `NotificationManagerCompat` instance

However, note that we have also defined an action on the primary `Notification`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
private NotificationCompat.Builder buildNormal() {
  NotificationCompat.Builder b=new NotificationCompat.Builder(this);

  b.setAutoCancel(true)
   .setDefaults(Notification.DEFAULT_ALL)
   .setContentTitle(getString(R.string.download_complete))
   .setContentText(getString(R.string.fun))
   .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
   .setSmallIcon(android.R.drawable.stat_sys_download_done)
   .setTicker(getString(R.string.download_complete))
   .addAction(android.R.drawable.ic_media_play,
      getString(R.string.play),
      buildPendingIntent(Settings.ACTION_SETTINGS));

  return(b);
}
```

`addAction()` on `WearableExtender` *replaces*, for the Wear device, any actions defined on the `Notification` itself using `addAction()`, but not the action defined via `setContentIntent()`.

On the primary device, we do not see the wear-only action:

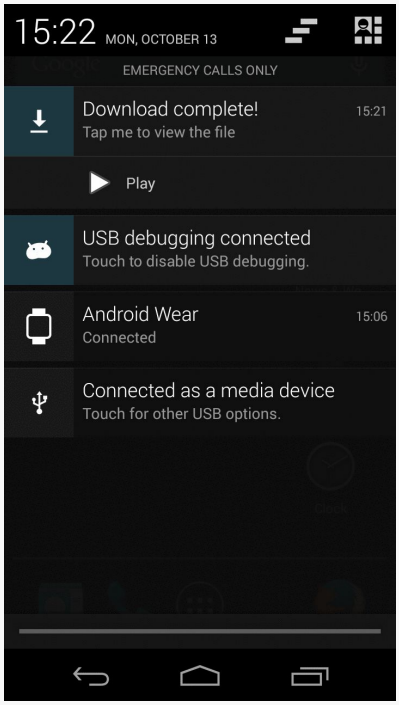


Figure 519: WearActions Demo, on a Galaxy Nexus

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

On a Wear device, though, we see both the wear-only and the main content action, but not the device-only action added via `addAction()` on the `NotificationCompat.Builder`:

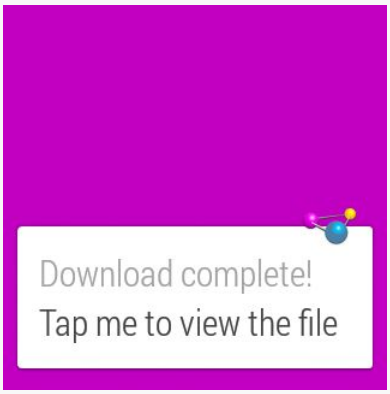


Figure 520: WearActions Demo, on a Samsung Galaxy Wear, Showing Notification

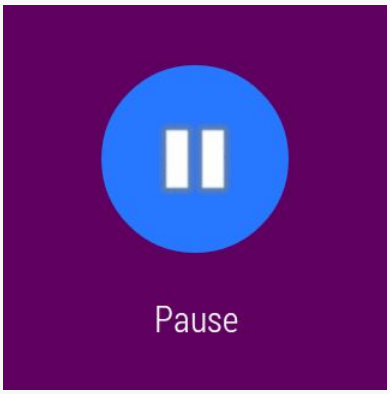


Figure 521: WearActions Demo, on a Samsung Galaxy Wear, Showing Wear-Only Action

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 522: WearActions Demo, on a Samsung Galaxy Wear, Showing Main Content Action

Hence:

- If you want actions only on the primary device, define those before applying a `WearableExtender` and its `addAction()`
- If you want actions only on the Wear device, define those using a `WearableExtender` and its `addAction()`
- If you want the same actions on both devices, define those using both flavors of `addAction()` (on `NotificationCompat.Builder` for the primary device and on `WearableExtender` for the Wear device)

Voice Input

In the spirit of Dick Tracy's two-way wrist radio, Android Wear allows you to talk to your wrist and not seem like you are completely insane.

In particular, your `Notification`, when presented on the Wear, can request that the user provide you with a response, via voice input or via canned responses. This can be very handy:

- Responding to a text message without pulling out one's phone
- Directing your app to file an incoming email message into a particular folder or label
- Responding to a police alert, requesting your assistance, by indicating that you will be on your way as soon as you can find your bright yellow trenchcoat
- And so on

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In many cases, with a regular `Notification`, the result of the user choosing an action is for us to display an activity. Sometimes, though, that's not what we want, such as a music player's `Notification` handling "pause" and similar events via its background service. Similarly, actions from a `Notification` seen on a Wear device will sometimes need to perform operations in the background, as the user may not be in position to look at your UI. This is especially true with voice input — usually, if we are bothering to dictate words to our wrist, that should happen instead of opening up the primary device. As a result, our flow for responding to the action is a little bit different, as is illustrated in the Notifications/VoiceInput sample project.

The Activity and Notification

Let's walk through the `MainActivity` that sets up our `Notification`:

```
package com.commonsware.android.wearvoice;

import android.app.Activity;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.support.v4.app.NotificationCompat;
import android.support.v4.app.NotificationManagerCompat;
import android.support.v4.app.RemoteInput;

public class MainActivity extends Activity {
  private static final int NOTIFY_ID=1337;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Intent i=new Intent(this, VoiceReceiver.class);
    PendingIntent pi=
        PendingIntent.getBroadcast(this, 0, i,
            PendingIntent.FLAG_UPDATE_CURRENT);

    RemoteInput remoteInput=
        new RemoteInput.Builder(VoiceReceiver.EXTRA_SPEECH)
          .setLabel(getString(R.string.talk))
          .setChoices(getResources().getStringArray(R.array.replies))
          .build();

    NotificationCompat.Action wearAction=
        new NotificationCompat.Action.Builder(
              android.R.drawable.ic_btn_speak_now,
              getString(R.string.talk),
              pi).addRemoteInput(remoteInput).build();

    NotificationCompat.WearableExtender wearExtender=
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        new NotificationCompat.WearableExtender()
            .addAction(wearAction);

    NotificationCompat.Builder builder=
        new NotificationCompat.Builder(this)
            .setSmallIcon(android.R.drawable.stat_sys_download_done)
            .setContentTitle(getString(R.string.title))
            .setContentText(getString(R.string.talk))
            .extend(wearExtender);

    NotificationManagerCompat
      .from(this)
      .notify(NOTIFY_ID, builder.build());

    finish();
  }
}
```

We start by creating a broadcast `PendingIntent`, pointing to a `VoiceReceiver` that will respond to the voice input. We will examine this `VoiceReceiver` later in this example.

We then set up a `RemoteInput.Builder`. This is a builder-style API for defining a `RemoteInput` configuration to attach to a Wear-only action. Here, we configure it with:

- the key for retrieving the response in our `VoiceReceiver` (`VoiceReceiver.EXTRA_SPEECH`)
- the label to prompt the user for what we are looking for them to provide (an `R.string.talk` string resource)
- a `String` array of canned responses that the user can choose from rather than dictate their own answer and go through speech-to-text conversion (pulled from an `R.array.replies` `<string-array>` resource)

That `RemoteInput` is then applied to a `NotificationCompat.Action`, via its `NotificationCompat.Action.Builder` and the `addRemoteInput()` method. That `Action`, in turn, is wrapped in a `NotificationCompat.WearableExtender`, which is used to `extend()` a `NotificationCompat.Builder`.

Finally, the resulting `Notification` is raised using a `NotificationManagerCompat` instance.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Receiver

Our `VoiceReceiver`, registered in the manifest, is set up to respond to the voice action:

```
package com.commonsware.android.wearvoice;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.support.v4.app.RemoteInput;
import android.util.Log;
import android.widget.TextView;

public class VoiceReceiver extends BroadcastReceiver {
  static final String EXTRA_SPEECH="speech";

  @Override
  public void onReceive(Context ctxt, Intent i) {
    Bundle input=RemoteInput.getResultsFromIntent(i);

    if (input!=null) {
      CharSequence speech=input.getCharSequence(EXTRA_SPEECH);

      if (speech!=null) {
        Log.d(getClass().getSimpleName(), speech.toString());
      }
      else {
        Log.e(getClass().getSimpleName(), "No voice response speech");
      }
    }
    else {
      Log.e(getClass().getSimpleName(), "No voice response Bundle");
    }
  }
}
```

It uses `RemoteInput.getResultsFromIntent(i)` to pick out the response we got from the user for this action. There are three major possibilities:

1. We did not get any response (should not happen)
2. We got a response, but for whatever reason, the decoded `Bundle` is missing our `VoiceReceiver.EXTRA_SPEECH` key (also should not happen)
3. The `CharSequence` from the `VoiceReceiver.EXTRA_SPEECH` key in the decoded `Bundle` is the user's response, whether from speech recognition or from choosing one of our canned responses

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In this case, we just log the message to LogCat, but in principle you could do whatever you wanted. Just bear in mind that your UI may not be in the foreground, and that the device screen may be off entirely. It is also possible that your process will have been terminated between the time you raised the `Notification` and the user got around to responding to it from the Wear device. Hence, you should be making few assumptions about the environment at the point when you get the voice response.

The Results

The Wear device starts off with a typical action:

Figure 523: VoiceInput Demo, on a Samsung Galaxy Wear, Showing Voice Action

Tapping it brings up a voice input screen, where the user can dictate some text:

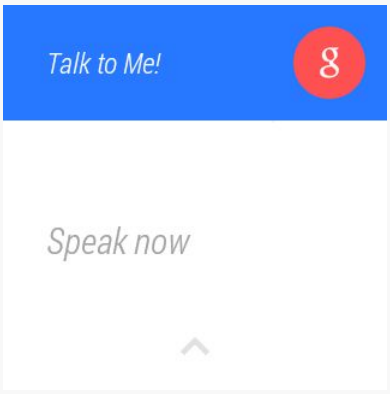


Figure 524: WearActions Demo, on a Samsung Galaxy Wear, Showing Voice Input

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If the user delays too long without saying anything recognizable, or if the user swipes up the screen, they are taken to our list of canned responses:

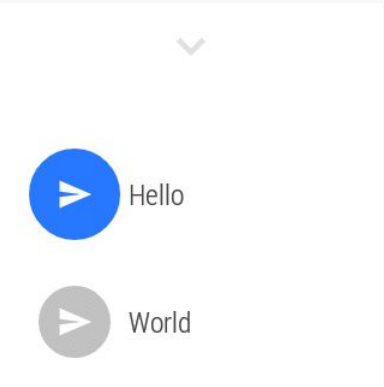


Figure 525: WearActions Demo, on a Samsung Galaxy Wear, Showing Canned Responses

If the user instead does dictate some text, initially they are shown just the interpreted text:

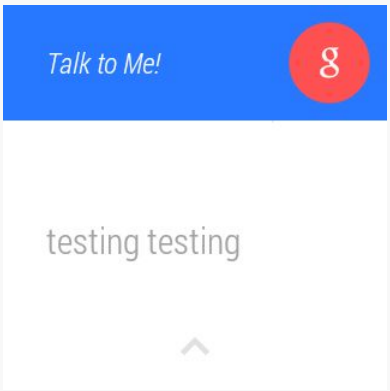


Figure 526: WearActions Demo, on a Samsung Galaxy Wear, Showing Voice Input Results

Then a cancel button with a progress indicator around the edge appears:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

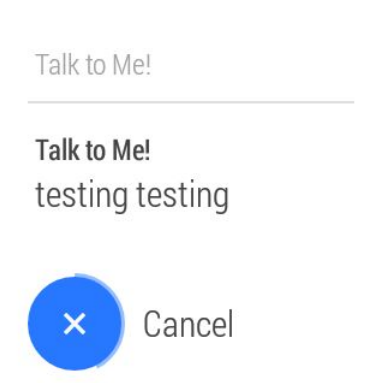


Figure 527: WearActions Demo, on a Samsung Galaxy Wear, Showing Voice Input Progress

If the user taps the cancel button before the progress indicator elapses, they are prompted to confirm or reject the input:

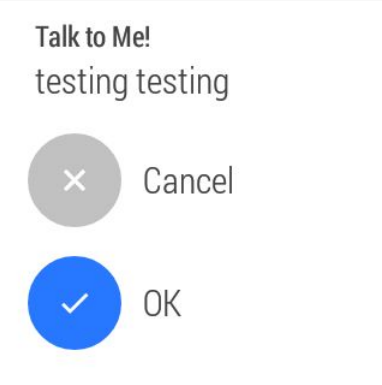


Figure 528: WearActions Demo, on a Samsung Galaxy Wear, Showing Voice Input Confirmation

Lockscreen Notifications

Historically, notification icons would be visible on the user's lockscreen, but that was it. This would give the user an indication of what apps need attention, but no additional context.

Android 5.0 now shows notifications on the lockscreen, to help provide that missing context. Now users can have more details about the notifications, to determine whether it is necessary to unlock the device right now to deal with them.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, this also raises privacy concerns, as now notification text can be seen by anyone with access to the phone. As such, Android 5.0 has introduced the concept of *visibility* to notifications, so developers can help control what is shown on the lockscreen versus what is shown only past the lockscreen.

However, these visibility options are only useful if:

- The device has a pattern, PIN, or password set, so it is not merely some swipe-to-unlock approach
- The user has indicated that the system should "hide sensitive notification content", either when they set up the pattern/PIN/password:

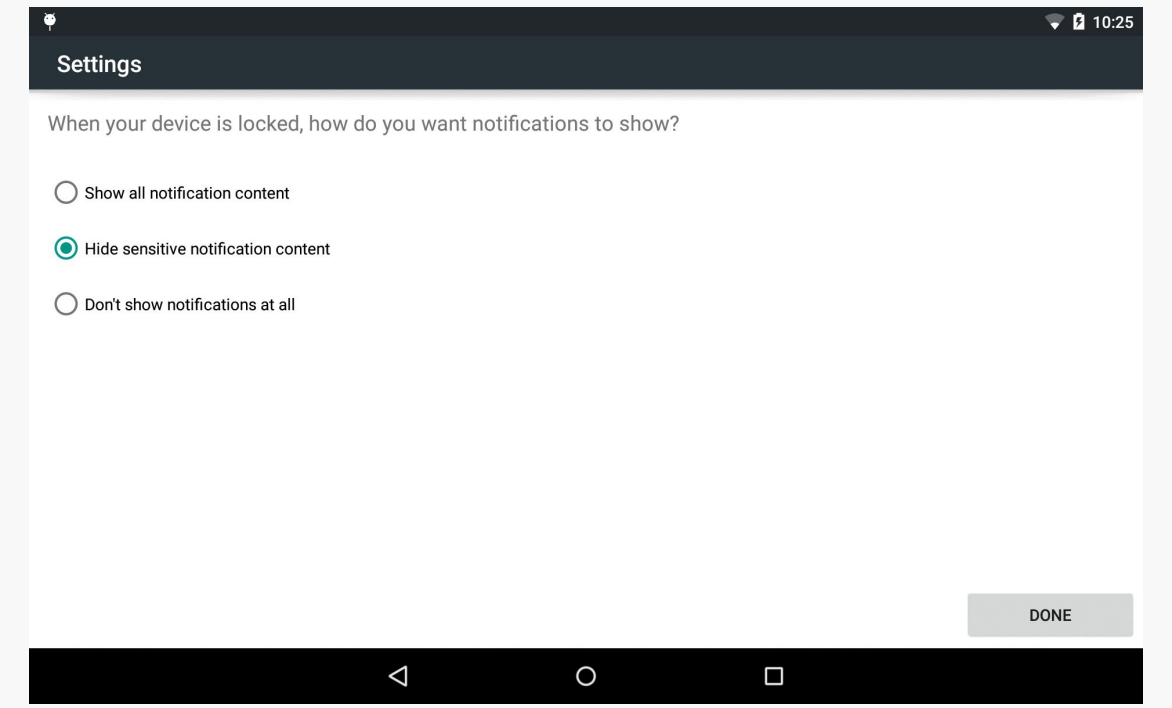


Figure 529: Choosing Notification Control, When Securing the Lockscreen

or in the "Sound & notification" portion of the Settings app:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

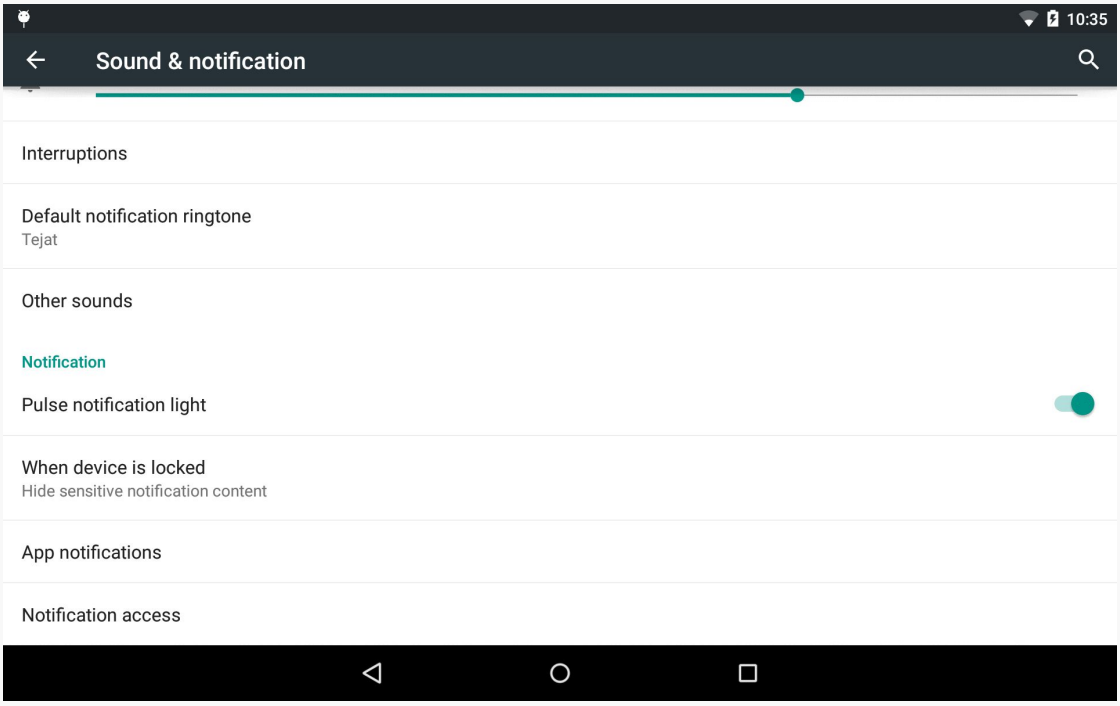


Figure 530: "Sound & notification" Settings

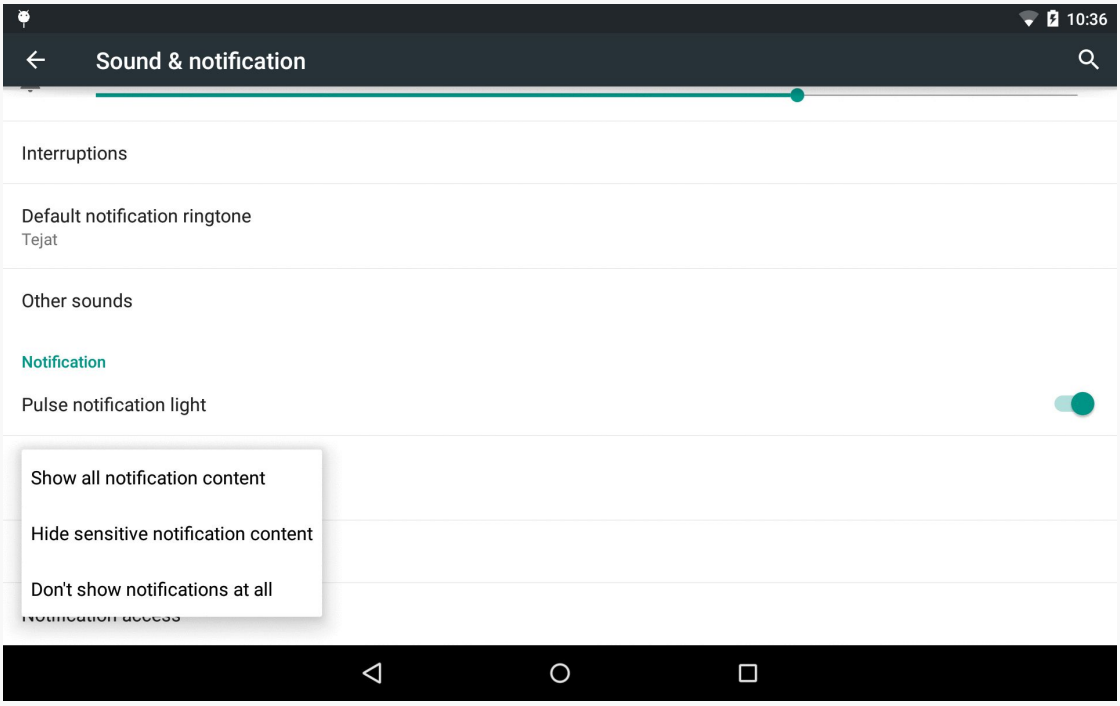


Figure 531: Notification Control Options in "Sound & notification" in Settings

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Given that the user has enabled "hide sensitive notification content" mode, you as a developer can choose a visibility to apply to your notifications. There are three such visibility options — private, public, and secret — covered in the following sections.

Private Notifications

The default behavior is a "private" `Notification`. Basic information appears on the lockscreen, but not the whole `Notification`. However, you as a developer can also provide a *separate* `Notification` that *will* be shown on the lockscreen, so you can choose what information appears publicly and what information does not.

The sample app for this section has a "public" edition of the `Notification` that shows up on the lockscreen:

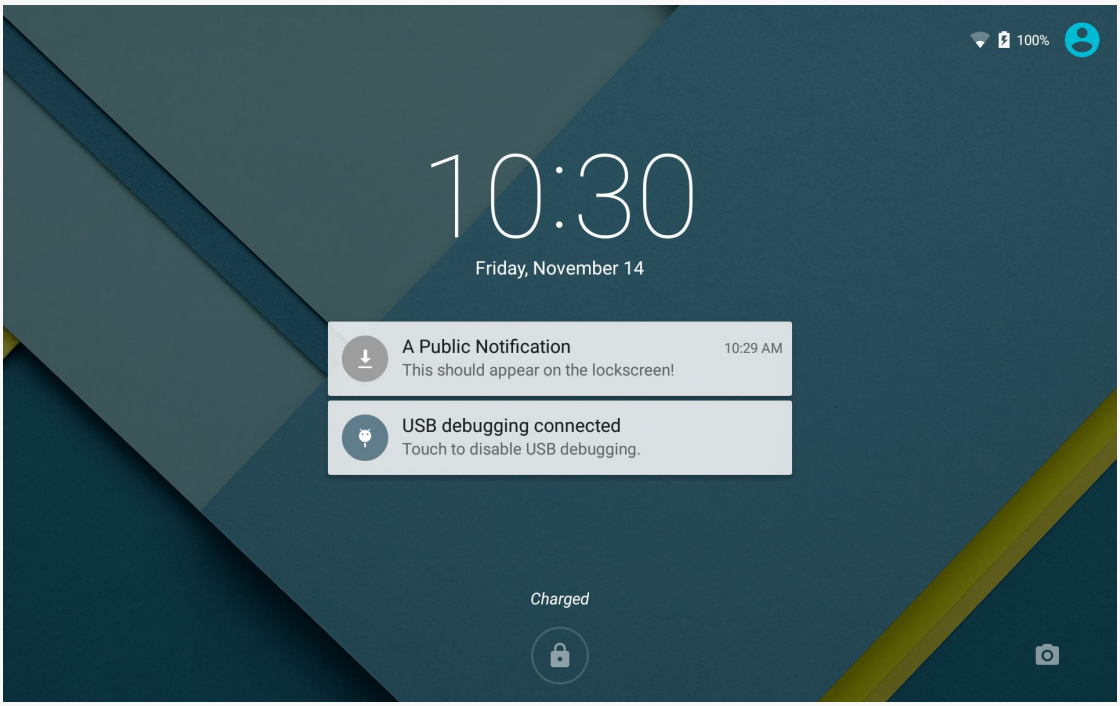


Figure 532: Public Edition of Private Lockscreen Notification, on a Nexus 7

Public Notifications

Instead of creating a separate `Notification` for public visibility on the lockscreen, you could flag your main `Notification` as having public visibility. This is suitable for notifications where there is little to no privacy implications for having the information appear on the lockscreen.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Secret Notifications

A `Notification` with visibility set to "secret" will not show up on the lockscreen at all. The ringtone, etc. will occur, as requested (and based on device settings, like it being muted), but otherwise there is no visible indication on the lockscreen that your `Notification` exists. Only when the user gets past the lockscreen will your `Notification` appear, in the status bar.

A Visibility Sample

The `Notifications/Lollipop` sample project demonstrates the use of these visibility values. It also demonstrates heads-up notifications, covered later in this chapter.

The user interface consists of a `Spinner` of possible `Notification` variants, a `SeekBar` to allow the user to specify a delay period in seconds before showing the `Notification`, and a `Button` to trigger showing the `Notification`:

```
<?xml version="1.0" encoding="utf-8"?>

<TableLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="8dp"
    android:stretchColumns="1">

    <TableRow>
        <TextView
            android:text="@string/type_label"/>

        <Spinner
            android:id="@+id/type"/>
    </TableRow>

    <TableRow>
        <TextView
            android:text="@string/delay_label"/>

        <SeekBar
            android:id="@+id/delay"
            android:progress="5"
            android:max="30"/>
    </TableRow>

    <Button
        android:text="@string/notify_button"
        android:id="@+id/download"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        android:onClick="notifyMe"/>
</TableLayout>
```

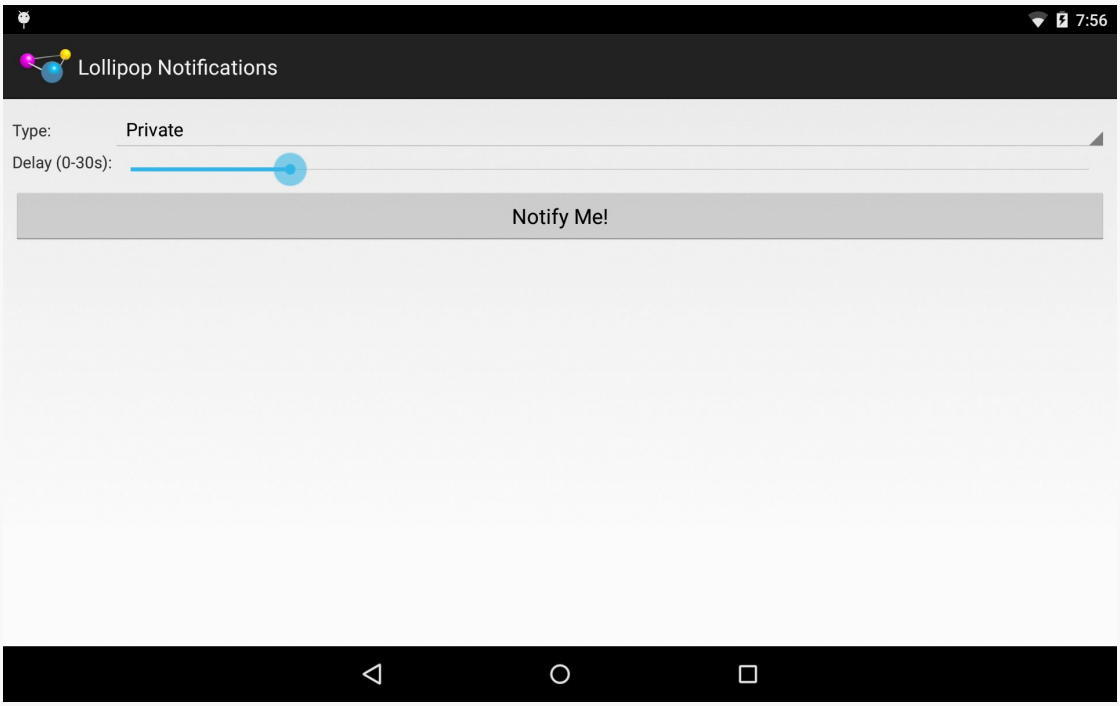


Figure 533: Lollipop Notifications Demo, on a Nexus 7

The `onCreate()` method of our launcher activity (`MainActivity`) initializes the UI:

```
package com.commonsware.android.lollipopnotify;

import android.app.Activity;
import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.SeekBar;
import android.widget.Spinner;

public class MainActivity extends Activity {
  private Spinner type=null;
  private SeekBar delay=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    type=(Spinner)findViewById(R.id.type);

    ArrayAdapter<String> types=
        new ArrayAdapter<String>(this,
            android.R.layout.simple_spinner_item,
            getResources().getStringArray(R.array.types));

    types.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
    type.setAdapter(types);

    delay=(SeekBar)findViewById(R.id.delay);
  }

  public void notifyMe(View v) {
    Intent i=new Intent(this, AlarmReceiver.class)
        .putExtra(AlarmReceiver.EXTRA_TYPE, type.getSelectedItemPosition());
    PendingIntent pi=PendingIntent.getBroadcast(this, 0, i,

PendingIntent.FLAG_UPDATE_CURRENT);
    AlarmManager mgr=(AlarmManager)getSystemService(ALARM_SERVICE);

    mgr.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime()+(1000*delay.getProgress()),
        pi);
  }
}
```

In particular, onCreate() populates the Spinner based on a <string-array> resource:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="types">
        <item>Private</item>
        <item>Public</item>
        <item>Secret</item>
        <item>Heads-Up</item>
    </string-array>
</resources>
```

When the button is clicked, the notifyMe() method on MainActivity is called. Here, we:

- Create an Intent pointing at an AlarmReceiver
- Package an extra on the Intent that contains the selected position of the Spinner
- Wrap the Intent in a getBroadcast() PendingIntent
- Use set() on AlarmManager to invoke the PendingIntent after the delay period specified via the SeekBar

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Since the `targetSdkVersion` of this project is below 19, the `set()` method will behave in an exact fashion, triggering our `AlarmReceiver` at the designated time.

`AlarmReceiver`, in turn, uses a `switch` statement to call out to different private methods based upon which `Spinner` item was selected:

```
package com.commonsware.android.lollipopnotify;

import android.app.Notification;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.provider.Settings;
import android.support.v4.app.NotificationCompat;
import android.support.v4.app.NotificationManagerCompat;

public class AlarmReceiver extends BroadcastReceiver {
  private static final int NOTIFY_ID=1337;
  static final String EXTRA_TYPE="type";

  @Override
  public void onReceive(Context ctxt, Intent i) {
    NotificationManagerCompat mgr=NotificationManagerCompat.from(ctxt);

    switch (i.getIntExtra(EXTRA_TYPE, -1)) {
      case 0:
        notifyPrivate(ctxt, mgr);
        break;

      case 1:
        notifyPublic(ctxt, mgr);
        break;

      case 2:
        notifySecret(ctxt, mgr);
        break;

      case 3:
        notifyHeadsUp(ctxt, mgr);
        break;
    }
  }

  private void notifyPrivate(Context ctxt, NotificationManagerCompat mgr) {
    Notification pub=buildPublic(ctxt).build();

    mgr.notify(NOTIFY_ID, buildNormal(ctxt).setPublicVersion(pub).build());
  }

  private void notifyPublic(Context ctxt, NotificationManagerCompat mgr) {
    mgr.notify(NOTIFY_ID,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        buildNormal(ctxt)
            .setVisibility(NotificationCompat.VISIBILITY_PUBLIC)
            .build());
  }

  private void notifySecret(Context ctxt, NotificationManagerCompat mgr) {
    mgr.notify(NOTIFY_ID,
        buildNormal(ctxt)
            .setVisibility(NotificationCompat.VISIBILITY_SECRET)
            .build());
  }

  private void notifyHeadsUp(Context ctxt, NotificationManagerCompat mgr) {
    mgr.notify(NOTIFY_ID,
        buildNormal(ctxt)
            .setPriority(NotificationCompat.PRIORITY_HIGH)
            .build());
  }

  private NotificationCompat.Builder buildNormal(Context ctxt) {
    NotificationCompat.Builder b=new NotificationCompat.Builder(ctxt);

    b.setAutoCancel(true)
        .setDefaults(Notification.DEFAULT_ALL)
        .setContentTitle(ctxt.getString(R.string.download_complete))
        .setContentText(ctxt.getString(R.string.fun))
        .setContentIntent(buildPendingIntent(ctxt,
Settings.ACTION_SECURITY_SETTINGS))
        .setSmallIcon(android.R.drawable.stat_sys_download_done)
        .setTicker(ctxt.getString(R.string.download_complete))
        .addAction(android.R.drawable.ic_media_play,
            ctxt.getString(R.string.play),
            buildPendingIntent(ctxt, Settings.ACTION_SETTINGS));

    return(b);
  }

  private NotificationCompat.Builder buildPublic(Context ctxt) {
    NotificationCompat.Builder b=new NotificationCompat.Builder(ctxt);

    b.setAutoCancel(true)
        .setDefaults(Notification.DEFAULT_ALL)
        .setContentTitle(ctxt.getString(R.string.public_title))
        .setContentText(ctxt.getString(R.string.public_text))
        .setContentIntent(buildPendingIntent(ctxt,
Settings.ACTION_SECURITY_SETTINGS))
        .setSmallIcon(android.R.drawable.stat_sys_download_done)
        .addAction(android.R.drawable.ic_media_play,
            ctxt.getString(R.string.play),
            buildPendingIntent(ctxt, Settings.ACTION_SETTINGS));

    return(b);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private PendingIntent buildPendingIntent(Context ctxt, String action) {
    Intent i=new Intent(action);

    return(PendingIntent.getActivity(ctxt, 0, i, 0));
  }
}
```

If the user chooses the "Private" option in the `Spinner`, we call `notifyPrivate()`. That method builds two `Notification` objects: the regular one and a separate public edition. We attach the public edition to the regular `Notification` via a call to `setPublicVersion()` on the `NotificationCompat.Builder`. Then, we raise the regular `Notification`. This will show the public edition if the lockscreen is locked; otherwise, it will show the regular edition.

If the user chooses the "Public" option, we call `notifyPublic()`. That, in turn, calls `setVisibility(NotificationCompat.VISIBILITY_PUBLIC)` on the `NotificationCompat.Builder`, causing our `Notification` to appear normally both on the lockscreen and past the lockscreen.

If the user chooses the "Secret" option, we call `notifySecret()`. That uses `setVisibility(NotificationCompat.VISIBILITY_SECRET)` to configure the `Notification` to only appear once the user has gotten past the lockscreen.

The "Heads-Up" option — fourth in the `Spinner` — is covered in the next section.

Priority, and Heads-Up Notifications

Notifications can have a priority associated with them. Normally, notifications with higher priority will appear higher in the list of notifications in the notification tray than will notifications with lower priority.

Android 5.0 takes this a step further, showing high-priority notifications in a "heads-up" style, popping up a small dialog-like window over the main screen, with the same basic content as would appear for the `Notification` in its tile in the notification tray:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

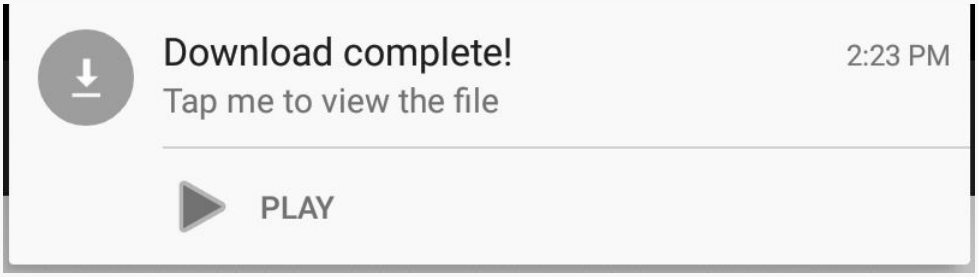


Figure 534: Lollipop Demo, on a Nexus 7, Showing Heads-Up Notification

Users can interact with the heads-up `Notification` or ignore it; in the latter case, the `Notification` will move into the status bar and the "heads-up" display will disappear from the screen.

Note that the "priority" concept being described here seems to be independent of the notion of "priority notifications" in the user's interruption configuration in Settings. There, "priority notifications" is tied to the app, not tied to any sort of configuration of the `Notification` itself.

Specifying the Priority

`NotificationCompat.Builder` has a `setPriority()` method that allows you to specify your requested priority. There are five priority values accepted as a parameter, all defined as constants out on the `NotificationCompat` class:

- `PRIORITY_MAX`
- `PRIORITY_HIGH`
- `PRIORITY_DEFAULT`
- `PRIORITY_LOW`
- `PRIORITY_MIN`

The actual priority applied to the `Notification` will depend upon other factors, and so you should not assume that your requested value will be accepted and applied as-is.

Results on Android 5.0 Devices

The heads-up `Notification` appears as shown in the above screenshot. The pop-up itself is centered across the top of the screen, as shown below:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

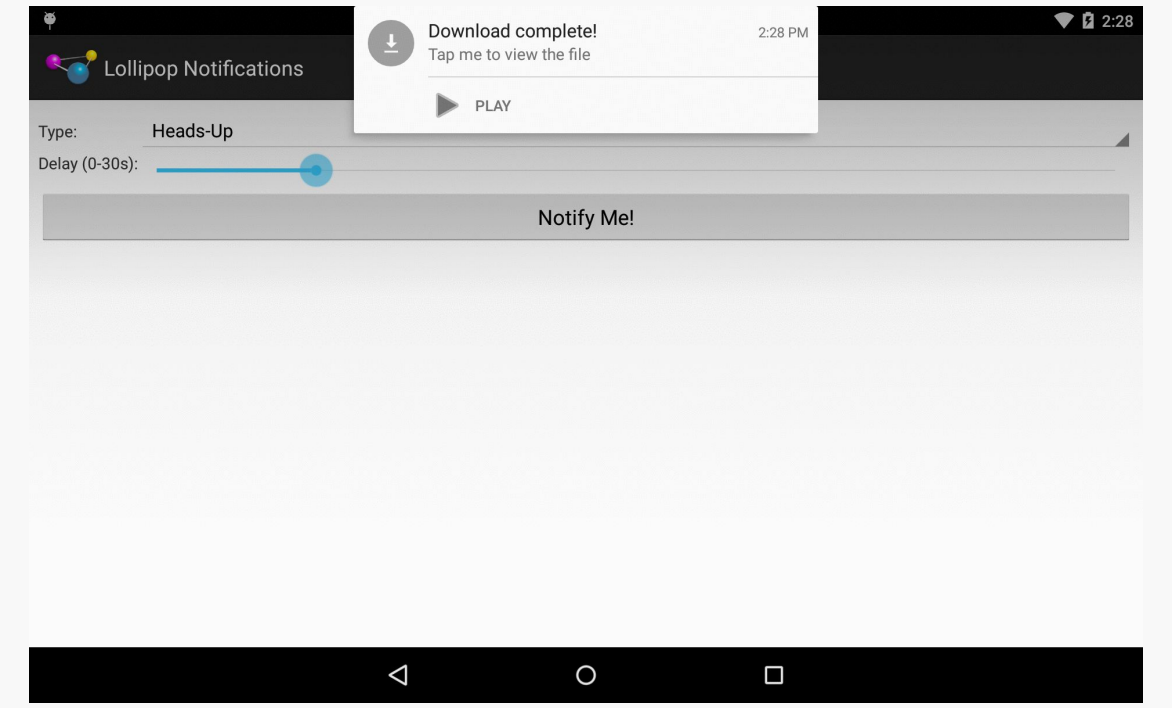


Figure 535: Lollipop Demo, on a Nexus 7, Showing Heads-Up Notification

After a few seconds of inactivity, the pop-up vanishes, and the `Notification` goes into the status bar.

Results on Older Devices

The concept of priority was introduced in API Level 16 (Android 4.1). On Android 4.1 through 4.4, the only effect of priority was to help influence the sort order of notifications in the notification tray, with higher-priority items drifting towards the top.

While `NotificationCompat.Builder` will allow you to specify a priority even on devices running older versions of Android than 4.1, the requested priority will be ignored, simply because priority did not exist back then. Hence, while your code will still work, it will have no effect on such old devices.

Full-Screen Notifications

Before Android 5.0 added heads-up notifications, while priority would influence things like sort order, it would have no real impact on how the user would be

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

informed about whatever event triggered the `Notification`. The user would still just get an icon in the status bar, and perhaps a ringtone and other hardware output.

However, sometimes we need to be somewhat more "in the user's face", such as for a calendar event reminder, or for an incoming phone call from our VOIP app.

It is tempting to launch an activity in these cases. In fact, that is what the user tends to perceive as happening, on Android 4.4 and older devices. And some apps no doubt actually do launch an activity.

A "middle ground" between showing a `Notification` and launching an activity is to use a full-screen `Notification`. Here, we provide a `PendingIntent` that should be invoked if the user is actively using the device at the time of the `Notification`. Typically, that `PendingIntent` will display an activity. However, on Android 5.0, the behavior has changed, where a full-screen `Notification` actually just triggers a heads-up notification, as would a high-priority `Notification`.

Requesting Full-Screen Output

All you need to do to set up a `Notification` to be full-screen is to call `setFullScreenIntent()` on your `NotificationCompat.Builder`, supplying two values:

1. A `PendingIntent` to be invoked when the notification is added to the screen
2. A `boolean`, where `true` indicates that even if the user has blocked notifications, you want this one to appear

For example, in the Notifications/FullScreen sample project, `MainActivity` shows a `Notification` constructed via the `buildNormal()` method:

```
  private NotificationCompat.Builder buildNormal() {
    NotificationCompat.Builder b=new NotificationCompat.Builder(this);

    b.setAutoCancel(true)
     .setDefaults(Notification.DEFAULT_ALL)
     .setContentTitle(getString(R.string.download_complete))
     .setContentText(getString(R.string.fun))
     .setContentIntent(buildPendingIntent(Settings.ACTION_SECURITY_SETTINGS))
     .setSmallIcon(android.R.drawable.stat_sys_download_done)
     .setTicker(getString(R.string.download_complete))
     .setFullScreenIntent(buildPendingIntent(Settings.ACTION_DATE_SETTINGS),
true)
     .addAction(android.R.drawable.ic_media_play,
         getString(R.string.play),
         buildPendingIntent(Settings.ACTION_SETTINGS));
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(b);
  }
```

Here, the `PendingIntent` is created using the same `buildPendingIntent()` method as before, this time opening up a distinct screen from the Settings app.

Results on Android 5.0 Devices

On Android 5.0, the "full screen" `Notification` appears as a heads-up `Notification`:

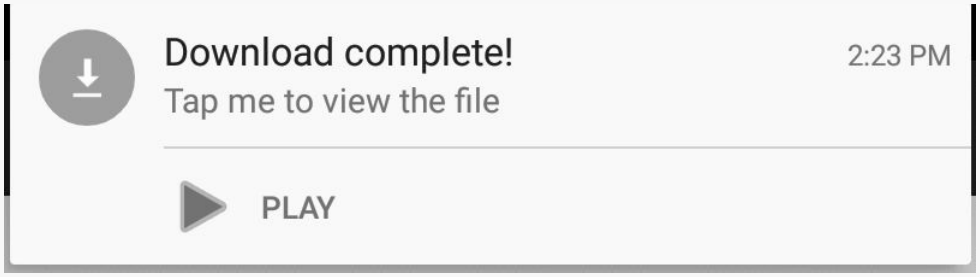


Figure 536: FullScreen Demo, on a Nexus 7, Showing "Full Screen" Notification

Note that there is no obvious way to actually invoke the `PendingIntent` associated with the `setFullScreenIntent()` method. Hence, you need to make sure that the `Notification` has some other means of getting the user to the right place in your UI, such as via `setContentIntent()` or an action.

Results on Android 3.0–4.4 Devices

On API Levels 11 through 19 (Android 3.0 through 4.4), the effect of a full-screen `PendingIntent` is to invoke the `PendingIntent` when the `Notification` is added to the screen. This will happen regardless of whether the user is using the device or not, though if the device is asleep, the activity triggered by the `PendingIntent` will only be visible once the user gets past their lockscreen.

Note that the `Notification` is *also* shown, along with whatever the `PendingIntent` does. That `Notification` is not automatically cleared when the user exits out of that activity via BACK, HOME, etc. Hence, it is up to you to clear that `Notification` if and when it is no longer relevant. The primary value of the `Notification` is to have the icon appear in the status bar on the lockscreen — even though the user cannot interact with your `Notification` then, the user may recognize your icon and therefore elect to unlock their device to see what all the fuss is about.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Results on Older Devices

Full-screen notifications were not supported prior to Android 3.0. While `NotificationCompat.Builder` will allow you to call `setFullScreenIntent()`, the value will be ignored prior to API Level 11.

In theory, there is nothing stopping `NotificationCompat` from launching an activity itself, in addition to displaying the `Notification`. However, at least at this time, it is not doing so, and it is fairly likely that Google will not add this in at this point.

Hence, the only way to do a "full-screen notification" is for your app to launch the desired activity, in addition to (or instead of) showing the `Notification`.

Custom Views: or How Those Progress Bars Work (Sometimes)

Some applications have specific tasks that take a chunk of time. The most common situation for these is a download — while downloading a file should not take forever, it may take several seconds or minutes, depending on the size of the file and the possible download speed.

You may have noticed that some applications, such as the Android Market, have a Notification for a download that shows the progress of the download itself. Visually, it's obvious how they accomplish this: they use a `ProgressBar`. But normally you create `Notification` objects with just a title and description as text. How do they get the `ProgressBar` in there? And, perhaps more importantly, how are they continuously updating it?

This section will explain how that works, along with a related construct added in Android 3.0: the custom ticker.

Custom Content

When you specify a title and a description for a `Notification`, you are implicitly telling Android to use a stock layout for the structure of the `Notification` object's entry in the notification drawer. However, instead, you can provide Android with the layout to use and the contents of all the widget, by means of a `RemoteViews`. In other words, by using the same techniques that you use to create app widgets, you can

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

create tailored notification drawer content. Just create the `RemoteViews` and put it in the `contentView` data member of the `Notification`.

To update the notification drawer content — such as updating a `ProgressBar` to show download progress — you update your `RemoteViews` in your `Notification` and re-raise the `Notification` via a call to `notify()`. Android will apply your revised `RemoteViews` to the notification drawer content, and the user will see the changed widgets. However, you will also want to remove requested features from the `Notification` that you do not want to occur every time you update the `RemoteViews`. For example, if you keep the `tickerText` in place, every time you update the `RemoteViews`, the ticker text will be re-displayed, which can get annoying.

We will see an example of this in action later in this chapter.

Custom Tickers

Traditionally, the "ticker" is a piece of text that is placed in the status bar when the `Notification` is raised, so that if the user happens to be looking at the phone at that moment (or glances at it quickly, cued by a vibration or ringtone), they get a bit more contextual information about the Notification and why it is there.

On API Level 11+ tablets, you also have the option of creating a custom ticker, once again using a `RemoteViews`. Create the `RemoteViews` to be what you want to show as the ticker, and assign it to the `tickerView` data member of the `Notification`. On devices with room (e.g., tablets), your `RemoteViews` will be displayed instead of the contents of the `tickerText` data member. However, it is a good idea to also fill in the `tickerText` value, for devices that elect to show that instead of your custom view.

Seeing It In Action

To see custom tickers and custom content in a complete project, take a peek at the `Notifications/HCNotifyDemo` sample project. This is perhaps the smallest possible project that uses all of these features, so do not expect much elaborate business logic.

The Activity

The launcher icon for this application is tied to an activity named `HCNotifyDemoActivity`. All it does is spawn a background service named

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

SillyService, that will simulate doing real work in the background and maintaining a Notification along the way:

```
package com.commonsware.android.hcnotify;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import com.commonsware.android.hcnotify.R;

public class HCNotifyDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    startService(new Intent(this, SillyService.class));

    finish();
  }
}
```

The IntentService

SillyService is an IntentService, to take advantage of the two key features of an IntentService: the supplied background thread, and automatically being destroyed when the work being done in the background is finished.

Since SillyService is an IntentService, and IntentService requires a constructor, supplying a display name for the service, we oblige:

```
  public SillyService() {
    super("SillyService");
  }
```

All of the rest of the business logic is in onHandleIntent(), which will be described in pieces below.

The Builder

In theory, SillyService is going to do some real long-running work, updating a ProgressBar in a Notification along the way. To keep the example simple — and not to violate "truth in advertising" laws given the service's name — SillyService will emulate doing real work by sleeping.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Hence, the first thing `SillyService` does in `onHandleIntent()` is get a `NotificationManager` and a `NotificationCompat.Builder`, then configure the builder to get the base `Notification` to use:

```
    NotificationManager
mgr=(NotificationManager)getSystemService(NOTIFICATION_SERVICE);
    NotificationCompat.Builder builder=new NotificationCompat.Builder(this);

    builder
      .setContent(buildContent(0))
      .setTicker(getText(R.string.ticker), buildTicker())
      .setContentIntent(buildContentIntent())
      .setLargeIcon(buildLargeIcon())
      .setSmallIcon(R.drawable.ic_stat_notif_small_icon)
      .setOngoing(true);

    Notification notif=builder.build();
```

Configuring the builder, in this case, involves calling the following setters:

1. `setContent()`, to provide the `RemoteViews` for the notification drawer entry, here delegated to a `buildContent()` method we will examine in a bit
2. `setTicker()`, to provide the material to be displayed as the ticker, in this case using a `setTicker()` variant that takes a `CharSequence` (e.g., a `String`, or the result of `getText()` on a string resource ID) and a `RemoteViews` to use in cases where the device supports custom tickers (delegated here to `buildTicker()`)
3. `setContentIntent()`, to provide the `PendingIntent` to be invoked if the user taps on our custom content `RemoteViews`, here delegated to `buildContentIntent()`
4. `setLargeIcon()`, used on some devices for a larger representation of our notification icon for use in tickers and non-custom notification drawer contents, here delegated to `buildLargeIcon()`
5. `setSmallIcon()`, use for the status bar/system bar icon and, on some devices, for non-custom notification drawer contents
6. `setOngoing()`, which sets `FLAG_ONGOING_EVENT`, preventing this `Notification` from being deleted by the user

Finally, we call `build()` to retrieve the `Notification` object as configured by the builder. Note that previous versions of `NotificationBuilder` used `getNotification()` instead of `build()`, but `getNotification()` is now officially deprecated.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The ProgressBar

Our `buildContent()` method just returns a `RemoteViews` object:

```
  private RemoteViews buildContent(int progress) {
    RemoteViews content=new RemoteViews(this.getPackageName(),
                                        R.layout.content);

    return(content);
  }
```

The `RemoteViews` object, in turn, is based on a trivial layout (`res/layout/content.xml`) containing a `ProgressBar`:

```
<ProgressBar xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/progress"
  style="?android:attr/progressBarStyleHorizontal"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:indeterminate="false">

</ProgressBar>
```

The simplest way to update a `ProgressBar` in a `Notification` is to simply hold onto the `Notification` object and update the `ProgressBar` in the `RemoteViews` as needed.

`SillyService` takes this approach, looping 20 times for 1000-millisecond naps, updating the `ProgressBar` on each pass of the loop:

```
    for (int i=0;i<20;i++) {
      notif.contentView.setProgressBar(android.R.id.progress,
                                       100, i*5, false);
      mgr.notify(NOTIFICATION_ID, notif);

      if (i==0) {
        notif.tickerText=null;
        notif.tickerView=null;
      }

      SystemClock.sleep(1000);
    }
```

You update the progress of a `ProgressBar` by calling `setProgressBar()` on the `RemoteViews`, and you get your content `RemoteViews` from the `contentView` data member of the configured `Notification`. `SillyService` has the `ProgressBar` run from 0 to 100 and sets the progress to be 5 times our loop counter. Each time we update the `RemoteViews`, we call `notify()` to raise or update the `Notification`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The key is that the first time we do this, we want to display our ticker, but not every time the `ProgressBar` updates, as that would really aggravate the user. So, after we raise the `Notification` in the first pass of our loop, we set the `tickerText` and `tickerView` data members of the `Notification` to `null`, to suppress further tickers from being displayed.

When the loop is finished, we just `cancel()` the `Notification`, to remove it from the screen.

The Rest of the Story

The `buildTicker()` method also returns a `RemoteViews`:

```
  private RemoteViews buildTicker() {
    RemoteViews ticker=new RemoteViews(this.getPackageName(),
                                       R.layout.ticker);

    ticker.setTextViewText(R.id.ticker_text,
                           getString(R.string.ticker));

    return(ticker);
  }
```

It, in turn, is based off of a `res/layout/ticker.xml` resource:

```
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/ticker_text"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="TextView">

</TextView>
```

There is nothing requiring the ticker (or the content, for that matter) to be completely static. You might well customize `TextView` or other widgets at runtime with details about the work being done. Here, `buildTicker()` does that via `setTextViewText()`, albeit just pulling in a string resource.

The `buildContentIntent()` method returns a `PendingIntent` to be invoked when the user taps on our `ProgressBar`-laden notification drawer entry. Here, lacking any better ideas and being generally lazy, we return a `PendingIntent` designed to bring up the Settings application:

```
  private PendingIntent buildContentIntent() {
    Intent i=new Intent(Settings.ACTION_SETTINGS);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(PendingIntent.getActivity(this, 0, i, 0));
  }
```

While small icons in a `Notification` must be resources, large icons are bitmaps. Presumably, that is to support the large icon holding contact photos, chat avatars, album art for music players, and whatnot. Hence, `buildLargeIcon()` needs to return a `Bitmap` object. In our case, it is simply a drawable resource, so we use `BitmapFactory` and `decodeResource()` to get a `Bitmap` from the PNG:

```
  private Bitmap buildLargeIcon() {
    Bitmap raw=BitmapFactory.decodeResource(getResources(),
                                            R.drawable.icon);

    return(raw);
  }
```

The Results

When we launch `HCNotifyDemoActivity`, which in turns starts up `SillyService`, we initially get our custom ticker on a tablet:

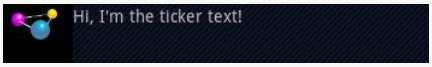


Figure 537: The custom ticker in our Notification, as seen on a Honeycomb tablet

Eventually, the ticker vanishes, leaving us with the traditional system bar icon:

Figure 538: The system bar icon for our Notification, as seen on a Honeycomb tablet

Tapping on the icon brings up the notification drawer, with our custom content, including our `ProgressBar`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

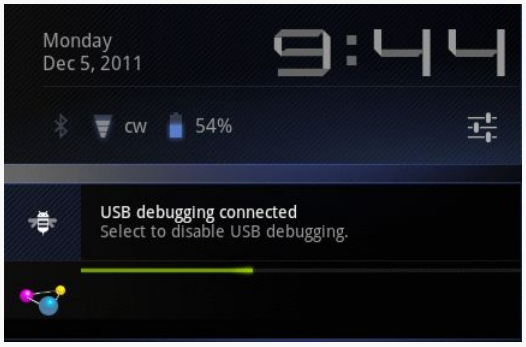


Figure 539: Notification ProgressBar, on Android 3.x Tablet

On an Android 4.0 phone, the status bar and ticker are no different than their Android 1.x/2.x counterparts, though we still get our custom content:

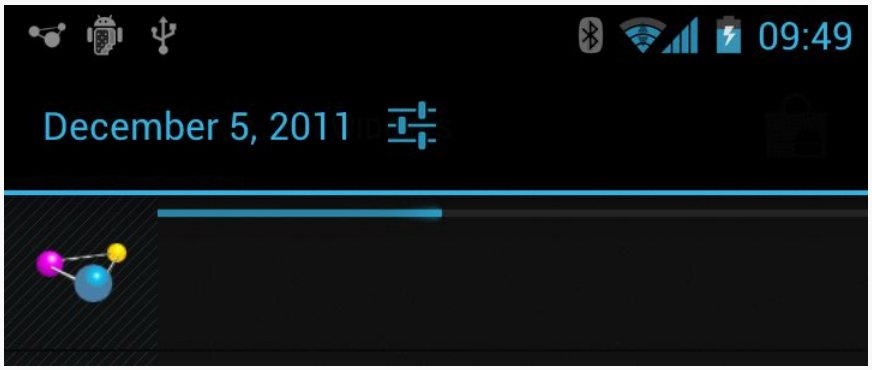


Figure 540: Notification ProgressBar, on Android 4.x Phone

How You *Really* Do Progress Notifications

While it is entirely possible to have a `Notification` with a `ProgressBar` using `RemoteViews`, there is a far simpler solution: use `setProgress()` on the `Notification.Builder` (or `NotificationCompat.Builder`). This is illustrated in the `Notifications/Progress` sample project, which is simplified clone of the `HCNotifyDemo` project.

The `SillyService` `onHandleIntent()` method now eschews the `RemoteViews` in favor of `setProgress()`:

```
@Override
protected void onHandleIntent(Intent intent) {
  NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    NotificationCompat.Builder builder=
        new NotificationCompat.Builder(this);

    builder.setTicker(getText(R.string.ticker))
           .setContentTitle(getString(R.string.progress_notification))
           .setContentText(getString(R.string.busy))
           .setContentIntent(buildContentIntent())
           .setSmallIcon(R.drawable.ic_stat_notif_small_icon)
           .setOngoing(true);

    for (int i=0; i < 20; i++) {
      builder.setProgress(20, i, false);
      mgr.notify(NOTIFICATION_ID, builder.build());

      SystemClock.sleep(1000);
    }

    builder.setContentText(getString(R.string.done))
           .setProgress(0, 0, false).setOngoing(false);

    mgr.notify(NOTIFICATION_ID, builder.build());
  }
```

After initializing the stock data for the `NotificationCompat.Builder`, we enter the same sort of loop–20-times-sleeping-a-second loop as was seen in the previous demo. This time, though, we call `setProgress()` on the `Builder` to set the current progress, then have it `build()` a `Notification` object to use with the `NotificationManager`.

`setProgress()` takes three parameters:

- the maximum value of the `ProgressBar`, which eventually routes to `setMax()` on the `ProgressBar` itself
- the current progress of the `ProgressBar`, which eventually routes to `setProgress()` on the `ProgressBar` itself
- `true` if the progress is indeterminate, `false` otherwise

In this case, we set the maximum to be 20 and the progress to be the value of our loop counter `i`.

When the loop is finished, we update the `Builder` to:

- Provide an alternative piece of text to show below the `ProgressBar`
- Remove the `ProgressBar` via `setProgress(0, 0, false)`
- Use `setOngoing(false)` to allow the user to get rid of the `Notification`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

When we run this sample, we see our `ProgressBar`-based `Notification` at the outset:

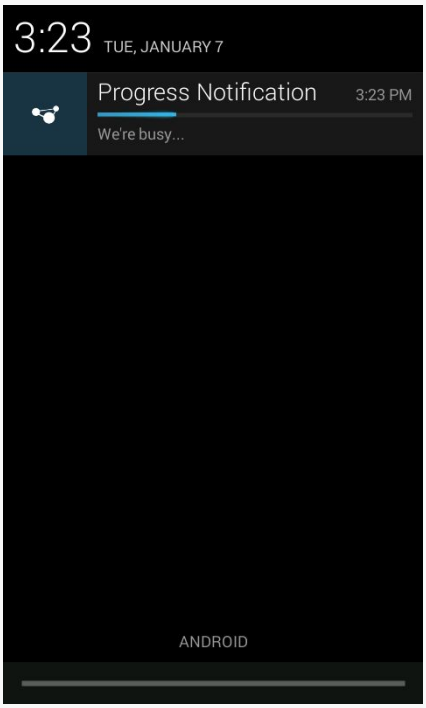


Figure 541: Notification with ProgressBar

When the loop is finished, we see the `Notification` as updated for its final state:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

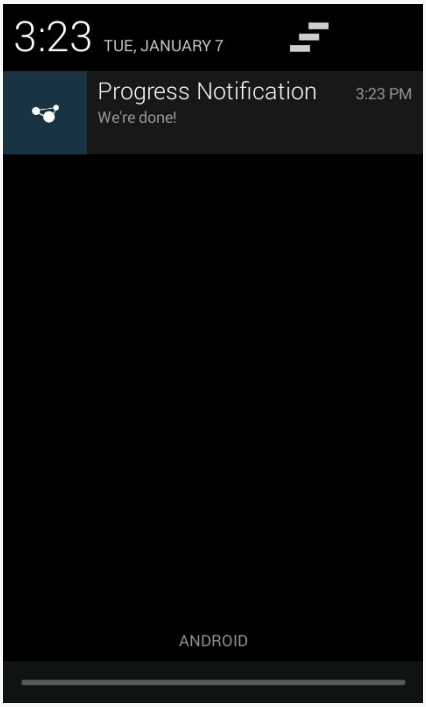


Figure 542: Notification with ProgressBar, When Done

Life After Delete

Most of the time, you do not care about your `Notification` being dismissed by the user from the notification drawer (e.g., pressing the Clear button on Android 1.x/2.x devices). If you do care about the `Notification` being deleted this way, you can supply a `PendingIntent` in the `deleteIntent` data member of the `Notification` — this will be executed when the user gets rid of your `Notification`. Usually, this will be a `getService()` or `getBroadcast()` `PendingIntent`, to have you do something in the background related to the dismissal. Users are likely to get rather irritated with you if you pop up an activity because they got rid of your `Notification`.

Note that this only works for `Notification` objects that can be cleared. If you have `FLAG_ONGOING_EVENT` set on the `Notification`, it will remain on-screen until *you* get rid of it.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Mysterious Case of the Missing Number

The `Notification` class has a `number` data member. On Android 1.x and 2.x, setting that data member would cause a number to be super-imposed on top of your icon in the status bar. That data member no longer works as of Android 3.0.

However, `Notification.Builder` has a `setNumber()` method which *does* work on API Level 11 and higher, though with slightly different behavior. Instead of putting the number on top of your status bar icon, the number will appear in your notification drawer entry. This only works if you do not use `setContent()` with `Notification.Builder` to define your own notification drawer entry layout — in that case, you could put your own number in wherever you would like.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

More Fun with Pagers

In earlier chapters, we saw basic uses of `ViewPager`, along with ways to show multiple pages at a time on larger screens. However, there are other ways to apply `ViewPager` and integrate it into the rest of your application, some of which we will examine in this chapter.

Prerequisites

This chapter assumes that you have read the core chapters, particularly the one showing how to use `ViewPager`.

Hosting ViewPager in a Fragment

Classically, the primary restriction on `ViewPager` was that you could not both have `ViewPager` be in a fragment *and* have `ViewPager` host fragments as its pages. You could do one or the other, but not both simultaneously.

As noted in a previous chapter, Android 4.2 supports nested fragments natively, and the latest Android Support package backport also supports nested fragments. With those, you can have `ViewPager` be in a fragment and host fragments as its pages. However, it requires a minor modification to the way we set up a `PagerAdapter`, as is illustrated in the `ViewPager/Nested` sample project. This is the same project as `ViewPager/Indicator`, with the twist that the pages are fragments and the `ViewPager` is inside a fragment.

Our activity now implements the standard add-the-fragment-if-it-does-not-exist pattern that we have seen previously:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.pagernested;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class ViewPagerIndicatorActivity extends FragmentActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getSupportFragmentManager().findFragmentById(android.R.id.content) ==
null) {
      getSupportFragmentManager().beginTransaction()
                                 .add(android.R.id.content,
                                      new PagerFragment()).commit();
    }
  }
}
```

This loads a `PagerFragment`, which contains most of the logic from our original activity:

```
package com.commonsware.android.pagernested;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.support.v4.view.PagerAdapter;
import android.support.v4.view.ViewPager;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class PagerFragment extends Fragment {
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.pager, container, false);
    ViewPager pager=(ViewPager)result.findViewById(R.id.pager);

    pager.setAdapter(buildAdapter());

    return(result);
  }

  private PagerAdapter buildAdapter() {
    return(new SampleAdapter(getActivity(), getChildFragmentManager()));
  }
}
```

The biggest difference is that our call to the constructor of `SampleAdapter` no longer uses `getSupportFragmentManager()`. Instead, it uses `getChildFragmentManager()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This allows `SampleAdapter` to use fragments hosted by `PagerFragment`, rather than ones hosted by the activity as a whole.

No other code changes are required, and from the user's standpoint, there is no visible difference.

Pages and the Action Bar

Fragments that are pages inside a `ViewPager` can participate in the action bar, supplying items to appear as toolbar buttons, in the overflow menu, etc. This is not significantly different than how any fragment participates in the action bar:

- Call `setHasOptionsMenu()` early in the fragment lifecycle (e.g., `onCreateView()`) to state that the fragment wishes to contribute to the action bar contents
- Override `onCreateOptionsMenu()` and `onOptionsItemSelected()`, much as you would with an activity

`ViewPager` and `FragmentManager` will manage the contents of the action bar, based upon the currently-visible page. That page's contributions will appear in the action bar, then will be removed when the user switches to some other page.

To see this in action, take a look at the `ViewPager/ActionBar` sample project. This is the same as the `ViewPager/Indicator` project from before, except:

- In `onCreateView()`, for even-numbered page positions (0, 2, etc.), we call `setHasOptionsMenu(true)`:

```
@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {
  View result=inflater.inflate(R.layout.editor, container, false);
  EditText editor=(EditText)result.findViewById(R.id.editor);

  position=getArguments().getInt(KEY_POSITION, -1);
  editor.setHint(getTitle(getActivity(), position));


  if ((position % 2)==0) {
    setHasOptionsMenu(true);
  }

  return(result);
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- In `onCreateOptionsMenu()`, we inflate a `res/menu/actions.xml` menu resource:

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  inflater.inflate(R.menu.actions, menu);

  super.onCreateOptionsMenu(menu, inflater);
}
```

Normally, we would also implement `onOptionsItemSelected()`, to find out when the action bar item was tapped, though this is skipped in this sample.

The result is that when we have an even-numbered page position — equating to an odd-numbered title and hint — we have items in the action bar:

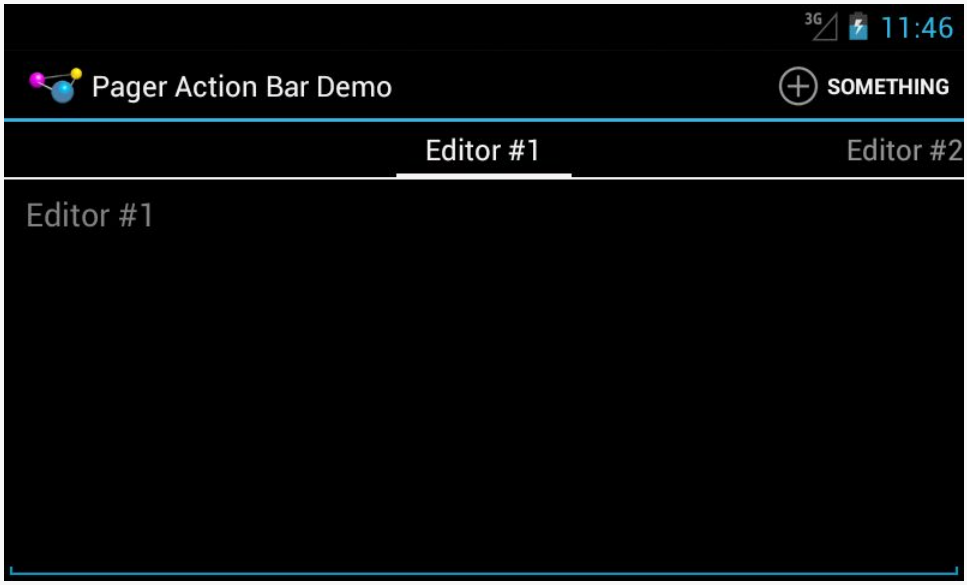


Figure 543: A ViewPager, PagerTabStrip, and Action Bar Item on Android 4.1

...but as soon as we swipe to an odd-numbered page position — equating to an even-numbered title and hint — our action bar item is removed, as that fragment did not call `setHasOptionsMenu(true)`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

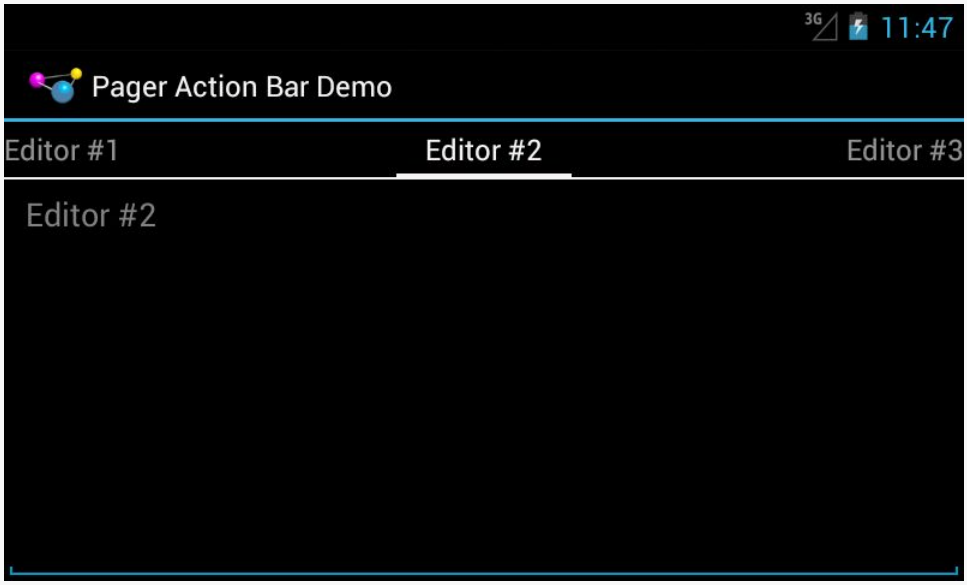


Figure 544: A ViewPager and PagerTabStrip, Sans Action Bar Item on Android 4.1

ViewPagers and Scrollable Contents

There are other things in Android that can be scrolled horizontally, besides a `ViewPager`:

- `HorizontalScrollView`
- `WebView`, for content that is wider than the width of the screen
- the deprecated `Gallery` widget
- maps from many mapping engines, such as Google Maps
- various third-party widgets

The challenge then comes in terms of dealing with horizontal swipe events. The ideal situation is for you to be able to swipe horizontally on the material inside the page, until you hit some edge (e.g., end of the `HorizontalScrollView`), then have swipe events move you to the adjacent page.

You can assist `ViewPager` in handling this scenario by subclassing it and overriding the `canScroll()` method. This will be called on a horizontal swipe, and it is up to you to indicate if the contents can be scrolled (returning `true`) or not (returning `false`). If the built-in logic is insufficient, tailoring `canScroll()` to your particular needs can help.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We will see an example of this later in the book, when we put some maps into a `ViewPager`.

Using ViewPagerIndicator

In addition to using `PagerTitleStrip` or `PagerTabStrip` to indicate context for the user, there are a plethora of other UI alternatives. Embodiments of many such patterns can be found in the ViewPagerIndicator library (VPI).

ViewPagerIndicator offers a variety of different indicator styles:

- A set of circles, colored lines, or your own icons to indicate different pages
- Another approach for implementing tabs, more readily customized for your needs (e.g., replacing the bar indicating the selected tab with an arrowhead)

ViewPagerIndicator is open source, released under the Apache Software License 2.0.

Downloading VPI

ViewPagerIndicator is distributed as an Android library project. On the ViewPagerIndicator home page you will find buttons to download a ZIP or a TGZ containing the library and sample code.

Android Studio users are a bit out of luck, in that there is no official AAR for this library. An unofficial AAR is available that you can use if you wish, using the following configuration of your top-level `repositories` and `dependencies` closures:

```
repositories {
    maven {
            url "http://dl.bintray.com/populov/maven"
    }
}

dependencies {
    compile 'com.viewpagerindicator:library:2.4.1@aar'
    compile 'com.android.support:support-v13:19.1.0'
}
```

The above example also shows loading in `support-v13`, as that (or `support-v4`) is needed for `ViewPager` itself.

If you are using Eclipse, you will need to import the project into your workspace, via File > Import, then choosing Android > Existing Android Code Into Workspace from

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the import wizard. Then you will need to attach the library to your main project, via Project > Properties > Android. This process is covered in greater detail in the chapter on library projects.

Replacing PagerTabStrip with TabPageIndicator

Earlier in the book, we looked at `PagerTabStrip`. The differences between using `PagerTabStrip` and one of the ViewPagerIndicator (VPI) classes, like `TabPageIndicator`, are fairly minor. The `ViewPager/VPI` sample project is a straight conversion of the `PagerTabStrip` sample to use `TabPageIndicator`.

First, your layout file will need to reference the VPI class instead of `PagerTabStrip`. The VPI indicators do not go as children of `ViewPager`; instead, you position them wherever makes sense using ordinary containers and layout rules. So, for example, you could have a classic vertical `LinearLayout` for stacking a `TabPageIndicator` atop the `ViewPager`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <com.viewpagerindicator.TabPageIndicator
    android:id="@+id/titles"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

  <android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  </android.support.v4.view.ViewPager>

</LinearLayout>
```

You also have to wire the indicator to the `ViewPager`, something that happens for you automatically when you use `PagerTabStrip`. This simply involves passing the `ViewPager` to the indicator via a `setViewPager()` method:

```
package com.commonsware.android.pager3;

import android.app.Activity;
import android.os.Bundle;
import android.support.v4.view.PagerAdapter;
import android.support.v4.view.ViewPager;
import com.viewpagerindicator.TabPageIndicator;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ViewPager pager=(ViewPager)findViewById(R.id.pager);
    TabPageIndicator tabs=(TabPageIndicator)findViewById(R.id.titles);

    pager.setAdapter(buildAdapter());
    tabs.setViewPager(pager);
  }

  private PagerAdapter buildAdapter() {
    return(new SampleAdapter(this, getFragmentManager()));
  }
}
```

For this sample, those are the only two required changes. However, if you were using an `OnPageChangeListener` with your `ViewPager`, you now need to attach it to the *indicator*, rather than the pager, so you find out about page changes triggered by the indicator. This is accomplished by calling `setOnPageChangeListener()` on your indicator object.

Styling the Indicator

It is also reasonably likely that you will want to style the indicator to meet your needs. In the case of the `TabPageIndicator`, you can control things like the font used for the tab title (size, color, style, etc.), the amount of padding to have to the left and right of the title to form the overall "tab", etc.

To do this, you will need to set up a custom theme for your project, if you do not already have one. That would involve creating a `style` resource, inheriting from whatever stock theme you want (e.g., `@android:style/Theme.Holo.Light.DarkActionBar`), and referencing your custom theme in your manifest:

```
  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name=".MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
  </activity>
</application>
```

In addition to any other theme customizations you want to make, you will need to have an `item`, named based upon the indicator class that you are using, that will point to a style resource to use for styling instances of that indicator class. In the case of `TabPageIndicator`, the name is `vpiTabPageIndicatorStyle`:

```
<style name="AppTheme" parent="@android:style/Theme.Holo.Light.DarkActionBar">
  <item name="vpiTabPageIndicatorStyle">@style/TabStyle</item>
</style>
```

Of course, you need to declare another `style` resource, using the name you supplied to the `vpiTabPageIndicatorStyle` `item`, where you tailor the look and feel of your indicator:

```
<style name="TabStyle" parent="Widget.TabPageIndicator">
  <item name="android:textColor">#FF33AA33</item>
  <item name="android:textSize">14sp</item>
  <item name="android:textStyle">italic</item>
  <item name="android:paddingLeft">16dp</item>
  <item name="android:paddingRight">16dp</item>
  <item name="android:fadingEdge">horizontal</item>
  <item name="android:fadingEdgeLength">8dp</item>
</style>
```

The easiest way to see what some of the options are is to look at the sample code for ViewPagerIndicator and the various custom styles that it defines.

With the custom style as defined above — specifying green italicized tab titles – we get tabs that look like these:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 545: ViewPager with Customized TabPagerIndicator

Columns for Large, Pages for Small

In some cases, you can take better advantage of larger screens by using `ViewPager` more judiciously. In a previous chapter, we explored having `ViewPager` itself display more than one page at a time. A variation on that same theme is to only use a `ViewPager` on screen sizes where you lack sufficient room for everything, and to put those same pages on the screen at the same time when you have room for all of them.

The Plume Example

Plume is a Twitter client for Android. It uses the columns-or-pages support for displaying various streams of tweets: your timeline, your @ mentions, hashtags you follow, etc.

Each stream is represented by a typical `ListView`, with one row per tweet. On a phone, since screen space is at a premium, those `ListView` widgets are set up in a `ViewPager`, with one list per page. Users can swipe between the lists, or use tabs to navigate the available lists.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, tablets offer more room, so they will show three `ListView` widgets side-by-side in landscape mode, so you can take in three sets of content without further interaction with the screen.

The `ViewPager/Columns1` sample project will demonstrate how you can accomplish the same basic approach in your own app... with some limitations. This is a clone of the ViewPagerIndicator sample from the previous section.

The Layouts

Our main activity layout — cunningly named `main` — has a `ViewPager`-based definition in `res/layout/main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <com.viewpagerindicator.TabPageIndicator
    android:id="@+id/titles"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

  <android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  </android.support.v4.view.ViewPager>

</LinearLayout>
```

However, in `res/layout-large/`, for 5-inch devices on up, we have a horizontal `LinearLayout` with three `FrameLayout` containers, each representing an equal-sized slot for one of our "pages":

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:baselineAligned="false"
  android:orientation="horizontal">

  <FrameLayout
    android:id="@+id/editor1"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <FrameLayout
    android:id="@+id/editor2"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1"/>

  <FrameLayout
    android:id="@+id/editor3"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1"/>

</LinearLayout>
```

Android will automatically inflate the proper layout when we call `setContentView(R.layout.main)`.

The Activity

However, while Android handles the inflation for us, we obviously need to populate the contents a bit differently. In this sample, though, we are relying upon the fact that screen size will not change on the fly. Hence, an instance of our application will either show a `ViewPager` *or* show the horizontal `LinearLayout`, and not have to switch between those at runtime.

Our `SampleAdapter`, therefore, can remain unchanged, except for reducing the page count to 3:

```
package com.commonsware.android.pagercolumns;

import android.app.Fragment;
import android.app.FragmentManager;
import android.content.Context;
import android.support.v13.app.FragmentPagerAdapter;

public class SampleAdapter extends FragmentPagerAdapter {
  Context ctxt=null;

  public SampleAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);
    this.ctxt=ctxt;
  }

  @Override
  public int getCount() {
    return(3);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public Fragment getItem(int position) {
    return(EditorFragment.newInstance(position));
  }

  @Override
  public String getPageTitle(int position) {
    return(EditorFragment.getTitle(ctxt, position));
  }
}
```

Our `MainActivity` will still use the `SampleAdapter`, and if we have a `ViewPager`, it will use it the same way as before. However, if we do *not* have a `ViewPager`, we must be showing three panes of content side by side, in which case we just execute a `FragmentTransaction` to populate the three `FrameLayout` containers with the three items created by the `SampleAdapter`:

```
package com.commonsware.android.pagercolumns;

import android.app.Activity;
import android.os.Bundle;
import android.support.v13.app.FragmentPagerAdapter;
import android.support.v4.view.ViewPager;
import com.viewpagerindicator.TabPageIndicator;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ViewPager pager=(ViewPager)findViewById(R.id.pager);

    if (pager == null) {
      if (getFragmentManager().findFragmentById(R.id.editor1) == null) {
        FragmentPagerAdapter adapter=buildAdapter();

        getFragmentManager().beginTransaction()
                                    .add(R.id.editor1,
                                         adapter.getItem(0))
                                    .add(R.id.editor2,
                                         adapter.getItem(1))
                                    .add(R.id.editor3,
                                         adapter.getItem(2)).commit();
      }
    }
    else {
      TabPageIndicator tabs=(TabPageIndicator)findViewById(R.id.titles);

      pager.setAdapter(buildAdapter());
      tabs.setViewPager(pager);
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }

  private FragmentPagerAdapter buildAdapter() {
    return(new SampleAdapter(this, getFragmentManager()));
  }
}
```

Of course, we skip the `FragmentTransaction` if the fragments already exist, such as due to a screen rotation configuration change.

The Results

On a phone, the `ViewPager`-based layout looks pretty much as it did before:

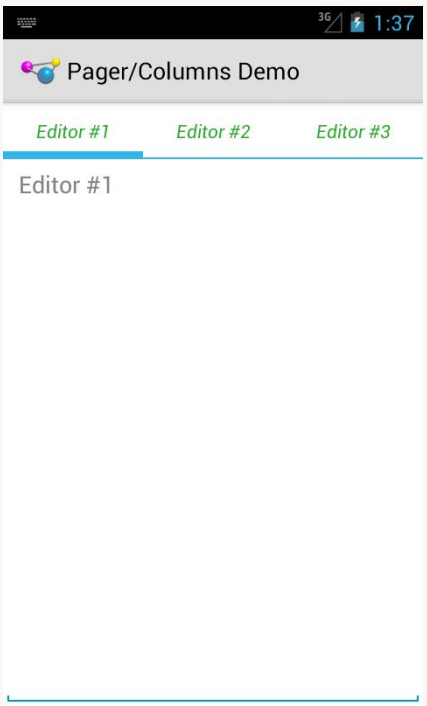


Figure 546: ViewPager with Customized TabPagerIndicator. Again.

However, on a tablet, we get our three editors side-by-side:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

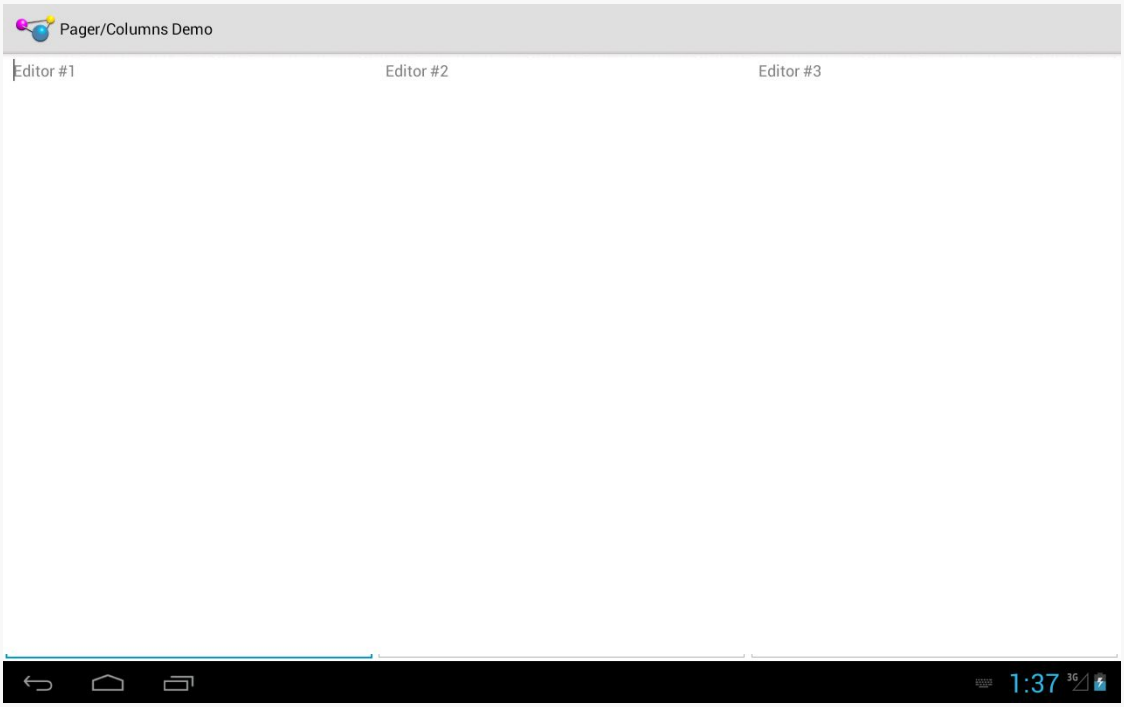


Figure 547: Same App, Large-Screen Layout with Side-By-Side Editors

The Limitations

The simplified large-screen layout does not contain any indicators above the three editors. This could be added by simple changes to the `res/layout-large/main.xml` layout resource, if desired.

The bigger limitation is that this only works if you want the same look in all configurations *except* screen size, and if the screen size never changes. However, it is eminently possible that you will want to have a different mix than that, such as using the three-column approach only on large-screen *landscape* layouts, using `ViewPager` everywhere else. In that case, our approach breaks down, as we will have different fragments inside the pager and outside the pager, meaning that we will lose our data on a configuration change. Addressing this issue is covered in the next two sections.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Introducing ArrayPagerAdapter

The flexibility of `ViewPager` is governed, to a large extent, by the implementation of its `PagerAdapter`. Inflexible `PagerAdapter` implementations lead to inflexible uses of `ViewPager`.

Notably, the two concrete `PagerAdapter` implementations shipped in the Android Support package — `FragmentPagerAdapter` and `FragmentStatePagerAdapter` — have their limitations when it comes to things like:

- Using fragments created by those adapters in other fashions, such as in the columns-or-pager scenario from the previous section
- Handling dynamically-changing contents, such as adding pages, removing pages, or reordering pages

The `ArrayPagerAdapter` is an attempt to provide a more flexible `PagerAdapter` implementation that still feels a lot like `FragmentPagerAdapter` in terms of its use of fragments. It also bears some resemblance to the `ArrayAdapter` used for `AdapterView` implementations like `ListView`, giving rise to its name.

`ArrayPagerAdapter` is part of the CWAC-Pager project and is available for use in any Android project compatible with the Apache License 2.0.

We will review the implementation of `ArrayPagerAdapter` later in this chapter. This section reviews how you can use `ArrayPagerAdapter` in your projects.

Adding the JAR

As the CWAC-Pager project does not need its own resources, it is packaged in the form of a simple JAR file.

Eclipse users can download the JAR and add to their projects by conventional means (e.g., putting it in the `libs/` directory). Android Studio users can use the `repositories` and `dependencies` closures outlined on the CWAC-Pager project home page:

```
repositories {
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
dependencies {
    compile 'com.commonsware.cwac:pager:0.2.+'
}
```

Choosing the Package

There are two implementations of `ArrayPagerAdapter`. One, in the `com.commonsware.cwac.pager` package, is designed for use with native API Level 11 fragments. The other, in the `com.commonsware.cwac.pager.v4` package, is designed for use with the Android Support package's backport of fragments. You will need to choose the right `ArrayPagerAdapter` for the type of fragments that you are using.

However, other than choosing suitable versions of classes for `Fragment`, etc., there is no real public API difference between the two. Hence, the documentation that follows is suitable for either implementation of `ArrayPagerAdapter`, so long as you use the one that matches the source of your fragment implementation.

Note that only `ArrayPagerAdapter` lives in the `com.commonsware.cwac.pager.v4` package. The classes and interfaces that support `ArrayPagerAdapter`, like `PageDescriptor`, are implemented in `com.commonsware.cwac.pager` and used by both implementations of `ArrayPagerAdapter`.

Creating PageDescriptors

You might think that `ArrayPagerAdapter` would take an array of pages, much like `ArrayAdapter` takes an array of models.

That's not how it works.

Instead, `ArrayPagerAdapter` wants an `ArrayList` of `PageDescriptor` objects. `PageDescriptor` is an interface, requiring you to supply implementations of two methods:

- `getTitle()`, which will be the title used for this page, for things like `PagerTabStrip` and the ViewPagerIndicator family of indicators
- `getFragmentTag()`, which is a unique tag for this page's fragment

Also, `PageDescriptor` extends the `Parcelable` interface, and so any implementation of `PageDescriptor` must also implement the methods and `CREATOR` required by `Parcelable`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You are welcome to create your own `PageDescriptor` if you wish. However, there is a built-in implementation, `SimplePageDescriptor`, which probably meets your needs. You just pass the tag and title into the `SimplePageDescriptor` constructor, and it handles everything else, including the `Parcelable` implementation.

Creating and Populating the Adapter

To work with `ArrayPagerAdapter`, you start by creating an `ArrayList` of `PageDescriptor` objects, one for each page that is to be in your pager.

Then, create a subclass of `ArrayPagerAdapter`. `ArrayPagerAdapter` uses Java generics, requiring you to declare the type of fragment the adapter is serving up to the `ViewPager`. So, for example, if you have a `ViewPager` that will have each page be an `EditorFragment`, you would declare your custom `ArrayPagerAdapter` like so:

```
  static class SamplePagerAdapter extends
```

If you will have pages come from a variety of fragments, just use the `Fragment` base class appropriate for your fragment source (e.g., `android.app.Fragment`).

Your custom `ArrayPagerAdapter` subclass will need to override (at minimum) one method: `createFragment()`. This method is responsible for instantiating fragments, as requested. You are passed the `PageDescriptor` for the fragment to be created — you simply create and return that fragment.

Hence, a custom `ArrayPagerAdapter` can be as simple as:

```
  static class SamplePagerAdapter extends
      ArrayPagerAdapter<EditorFragment> {
    public SamplePagerAdapter(FragmentManager fragmentManager,
                              ArrayList<PageDescriptor> descriptors) {
      super(fragmentManager, descriptors);
    }

    @Override
    protected EditorFragment createFragment(PageDescriptor desc) {
      return(EditorFragment.newInstance(desc.getTitle()));
    }
```

Then, you can create an instance of your custom `ArrayPagerAdapter` subclass as needed, supplying the constructor with a suitable `FragmentManager` and your `ArrayList` of `PageDescriptor` objects. Once attached to a `ViewPager`, `ArrayPagerAdapter` behaves much like a `FragmentPagerAdapter` by default.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

There is another flavor of the `ArrayPagerAdapter` constructor, one that takes a `RetentionStrategy` as a parameter, as is described later in this chapter.

Modifying the Contents

`ArrayPagerAdapter` offers several methods to allow you to change the contents of the `ViewPager`:

- `add()` takes a `PageDescriptor` and adds a new page at the end of the current roster of pages
- `insert()` takes a `PageDescriptor` and an insertion point and inserts a new page before the current page at that insertion point
- `remove()` takes a position and removes the page at that position
- `move()` takes an old and new position and moves the page from the old position to the new position (effectively combining a `remove()` from the old position and an `insert()` of the same page into the new position

Other Useful Methods

`getExistingFragment()`, given a position, returns the existing fragment for that position in the `ViewPager`, if that fragment exists. Otherwise, it returns `null`.

`getCurrentFragment()` is like `getExistingFragment()`, but returns the fragment for the currently-viewed page in the `ViewPager`.

Columns for Large *Landscape*, Pages for the Rest

Earlier in this chapter, we saw how you could conditionally use a `ViewPager` in some circumstances, but not others, such as using a `ViewPager` on smaller screens and a set of columns for the "pages" on larger screens. The limitation noted at that time was that you were stuck with one pattern for the lifetime of the activity, meaning that in any configuration change, you had to stick with the `ViewPager` or the columns that you started with.

However, while the columnar approach for larger screens works well in landscape, you may find the columns to be too tall and too skinny in portrait. Hence, a better solution would be to use columns only on larger screens in landscape, and to use the `ViewPager` everywhere else.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This is annoyingly tricky to do, assuming that you want to use the same fragments in each case, so you can arrange to hold onto the contents of their widgets.

Jake Wharton — author of ViewPagerIndicator and a seemingly infinite number of other Android open source libraries — raised this issue in a Google+ post. He also posted a sample solution, but one that was limited to only two fragments. Quoting Mr. Wharton:

> Due the shenanigans performed by FragmentPagerAdapter we're forced to write a custom PagerAdapter which handles the instances our selves.

However, while two pages is reasonable, having some flexibility for a few more pages would be useful. So, let's see how we can accomplish the same aims, using `ArrayPagerAdapter`, in the `ViewPager/FlexColumns` sample project.

Fragments Inside and Outside the ViewPager

A fragment cannot be in two containers at once. The `ViewPager`, where we have one, is a different container than one of our columns, when we have one.

Hence, if the container is not changing during the operation of our activity — such as using a `ViewPager` in both portrait and landscape on smaller screens — we have no problem. But, if the container *is* changing — such as switching between columns and a `ViewPager` on larger screens — we need to take steps.

One option for those "steps", of course, is to simply run a separate set of fragments. One set serves as pages of the `ViewPager`; the other serves as the columns. However, then we have to do work to synchronize those on configuration changes, as from the *user's* perspective, the fact that we happen to render things in pages or columns should not cause the user to lose data they entered in one form when switching to the other.

If we want to use the same fragment instances, then we can use normal configuration-change logic, like `onSaveInstanceState()`, to ensure that we hold onto user-entered data during the change. However, we have to arrange to move the fragment from one container to another. This will involve running a `FragmentTransaction` to `remove()` the fragment from the old container and `add()` it to its new container.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Making this more complicated is that the `PagerAdapter` should be handling the `add()` part, when the fragment is being put into a page, as that is how fragment-based `PagerAdapter` implementations like `FragmentPagerAdapter` work.

Adding to the fun is a matter of timing. By default, a `FragmentTransaction` is committed asynchronously. Attempting to `remove()` a fragment and `add()` the same fragment in the same transaction will fail, because the `add()` will complain that the fragment is already in another container, because the `remove()` will not have happened. Even doing the `remove()` and `add()` in separate normal transactions will not help. Instead, we need to ensure that the `remove()` has completed processing first, before we try to `add()`. To help with this, `FragmentManager` has a `executePendingTransactions()` method we can call, to have it complete its own processing on committed `FragmentTransactions` synchronously. Committing the `remove()` transaction and calling `executePendingTransactions()` before committing the `add()` transaction works.

The Revised PagerAdapter

With all that in mind, let's look at how this revised sample behaves. The core functionality is the same as with the earlier pager-or-columns sample, but now we will only use the columns on `-large` screen devices in `-land` orientation, by simply renaming `res/layout-large/` to `res/layout-large-land/`.

Our `PagerAdapter` is still called `SamplePagerAdapter`, but this time it is a `ArrayPagerAdapter` for our `EditorFragment` pages:

```
  static class SamplePagerAdapter extends
      ArrayPagerAdapter<EditorFragment> {
    public SamplePagerAdapter(FragmentManager fragmentManager,
                              ArrayList<PageDescriptor> descriptors) {
      super(fragmentManager, descriptors);
    }

    @Override
    protected EditorFragment createFragment(PageDescriptor desc) {
      return(createFragment(desc.getTitle()));
    }

    EditorFragment createFragment(String title) {
      return(EditorFragment.newInstance(title));
    }
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Revised Activity

The `onCreate()` method of the earlier example would see if we had a `ViewPager`, then either populate the columns or populate the `ViewPager` from our `PagerAdapter`. The `onCreate()` method of the new example does the same basic thing, except that it delegates most of the work for actually filling in the columns to a private `populateColumn()` method:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  ViewPager pager=(ViewPager)findViewById(R.id.pager);

  if (pager == null) {
    if (getFragmentManager().findFragmentById(R.id.editor1) == null) {
      SamplePagerAdapter adapter=buildAdapter();
      FragmentTransaction ft=
          getFragmentManager().beginTransaction();

      populateColumn(getFragmentManager(), ft, adapter, 0,
                     R.id.editor1);
      populateColumn(getFragmentManager(), ft, adapter, 1,
                     R.id.editor2);
      populateColumn(getFragmentManager(), ft, adapter, 2,
                     R.id.editor3);
      ft.commit();
    }
  }
  else {
    SamplePagerAdapter adapter=buildAdapter();
    TabPageIndicator tabs=(TabPageIndicator)findViewById(R.id.titles);

    pager.setAdapter(adapter);
    tabs.setViewPager(pager);
  }
}
```

The `buildAdapter()` method changes a bit, to create our `ArrayPagerAdapter` subclass using an array of `SimplePageDescriptor` objects:

```
private SamplePagerAdapter buildAdapter() {
  ArrayList<PageDescriptor> pages=new ArrayList<PageDescriptor>();

  for (int i=0; i < 3; i++) {
    pages.add(new SimplePageDescriptor(buildTag(i), buildTitle(i)));
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(new SamplePagerAdapter(getFragmentManager(), pages));
  }
```

`buildAdapter()`, in turn, uses `buildTag()` and `buildTitle()` methods to retrieve the tag and title to use given a position:

```
  private String buildTag(int position) {
    return("editor" + String.valueOf(position));
  }

  private String buildTitle(int position) {
    return(String.format(getString(R.string.hint), position + 1));
  }
```

Finally, our `populateColumn()` method handles the work to fill in one of our columns, if we are in column mode:

```
  private void populateColumn(FragmentManager fm,
                              FragmentTransaction ft,
                              SamplePagerAdapter adapter, int position,
                              int slot) {
    EditorFragment f=adapter.getExistingFragment(position);

    if (f == null) {
      f=adapter.createFragment(buildTitle(position));
    }
    else {
      fm.beginTransaction().remove(f).commit();
      fm.executePendingTransactions();
    }

    ft.add(slot, f, buildTag(position));
  }
```

First, we ask our `ArrayPagerAdapter` to retrieve for us the existing fragment, if any, for this given column/page, based on its position. This may return `null`, if this is the first time we have run our app, in which case we ask our `ArrayPagerAdapter` to create the fragment for us (using the same logic that it would when functioning inside of a `ViewPager`, via `createFragment()`).

Otherwise, `getExistingFragment()` should return an existing `EditorFragment` instance, one probably formerly managed by a `ViewPager`. So, we create, commit, and execute a `FragmentTransaction` to `remove()` this fragment from its existing container.

The net is that, in either case, we have an `EditorFragment`, set up for use in this column, that does not have a current container. To add it to our column, we simply

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

call `add()` on the supplied `FragmentTransaction`, which is committed by our activity's `onCreate()` method. However, we use the three-parameter form of `add()`, which allows us not only to put the fragment into a container, but to assign it a tag as well. The tag is how `ArrayPagerAdapter` identifies the various fragments — by using the same tag, this fragment can be picked up by future instances of `ArrayPagerAdapter` in case of a configuration change.

You will notice that while we `remove()` the `EditorFragment` from the `ViewPager` and `add()` it to the column, we are not handling the reverse case, where we would `remove()` the fragment from the column and/or `add()` it to the `ViewPager`. That little bit of logic is supplied to us by `ArrayPagerAdapter`, as we will see when we examine the implementation of `ArrayPagerAdapter` later in this chapter.

The resulting activity works exactly the same as the previous one, except that we use the `ViewPager` in portrait mode on larger-screen devices. Rotating a large-screen device will show our fragments moving between pages and columns, with their contents (whatever you type into the `EditorFragment` instances) being maintained via the built-in `onSaveInstanceState()` support for `EditText` widgets.

Adding, Removing, and Moving Pages

`ArrayPagerAdapter` also supports modifying the roster of pages at runtime: adding, inserting, removing, and moving pages. For example, a Twitter client might:

- Allow users to add pages for new monitored hashtags or search results
- Allow users to reorder the pages, putting more frequently-used ones towards the "front", for easier access when the app starts from scratch
- Allow users to remove pages they do not use, such as ones they added earlier

To see how this works in practice, we can examine the demo project for the CWAC-Pager library. There are two versions of this demo, one for the "v4" fragments from the Android Support package, and one for native API Level 11 fragments. Here, we will take a look at the latter project.

Reviewing the Core Functionality

This project is yet another rendition of our bunch-of-`EditorFragment`-pages sample that we have been examining for various ways of using `ViewPager`. This one sets up 10 pages at the start. However, it also inflates a menu resource to add four actions to the action bar: add, split, remove, and swap:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 548: ArrayPagerAdapter Demo App, Showing First 3 Pages and Action Bar

`onOptionsItemSelected()` in our activity routes those four action items to three methods: `add()` (for add and split), `remove()`, and `swap()`.

Add and Split

Tapping the "add" action bar item will add a new page before the current one, with a title and hint based upon the number of existing pages (e.g., tapping "add" with 10 pages will add "Editor #11"):

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 549: ArrayPagerAdapter Demo App, Showing Result of "Add" From Second Page

Tapping the "split" action bar item will add a new page after the currently-selected one.

Since both of these involve adding pages, this sample consolidates their work into a single `add()` method, taking a `boolean` parameter to indicate if we are inserting a page before the current one or after:

```
  private void add(boolean before) {
    int current=pager.getCurrentItem();
    SimplePageDescriptor desc=
        new SimplePageDescriptor(buildTag(adapter.getCount()),
                                 buildTitle(adapter.getCount()));

    if (before) {
      adapter.insert(desc, current);
    }
    else {
      if (current < adapter.getCount() - 1) {
        adapter.insert(desc, current + 1);
      }
      else {
        adapter.add(desc);
      }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }
```

We call `getCurrentItem()` on the `ViewPager` to determine what the position index is of the currently-selected page. From there, we set up our `SimplePageDescriptor` for the page that we will be adding, giving it a title based upon our `hint` string resource and a tag based upon the number of pages. We then call `add()` (if we are on the last page and the user clicked on "split") or `insert()` (for all other scenarios) to inject the new page. The `ArrayPagerAdapter` will be responsible for creating this page, just as it did for all previous pages.

Remove

Tapping "remove" will remove the currently-selected page, so long as we will still have at least one page remaining (just to keep the example simpler, so we do not have to worry about not having a "current page").

This is handled by the `remove()` method on our activity, which turns around and calls `remove()` on the `ArrayPagerAdapter`:

```
  private void remove() {
    if (adapter.getCount() > 1) {
      adapter.remove(pager.getCurrentItem());
    }
  }
```

Swap

Tapping "swap" will swap the positions of the current page and the one immediately after it. The exception is if you are on the last page, in which case we will swap the current page with the one immediately before it:

```
  private void swap() {
    int current=pager.getCurrentItem();

    if (current < adapter.getCount() - 1) {
      adapter.move(current, current + 1);
    }
    else {
      adapter.move(current, current - 1);
    }
  }
```

This is handled by the `swap()` method on our activity, which calls `move()` on the `ArrayPagerAdapter`. `move()` takes the position of the page to be moved and the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

position it should wind up in after the move, so we call `move(current, current + 1)` to swap the `current` page with the one after it or `move(current, current - 1)` to swap the `current` page with the one before it.

Inside ArrayPagerAdapter

`ArrayPagerAdapter` is a relatively large implementation of the `PagerAdapter` interface, and it helps to demonstrate some of the challenges faced when trying to create alternative fragment-based `PagerAdapter` implementations. Hence, this section will dive into portions of the innards of `ArrayPagerAdapter`, to explain how (and, sometimes, why) it does what it does.

Note that `ArrayPagerAdapter` will continue to expand over time, and so the copy in the master branch of the GitHub repo may be newer than the one profiled in this chapter. This chapter covers v0.1.1.

Also note that some of the code in `ArrayPagerAdapter` comes from `FragmentPagerAdapter` — as little of this code was altered as was practical, to help make it easier to integrate changes made to `FragmentPagerAdapter` over time.

Also, to simplify the discussion, this section will demonstrate the `ArrayPagerAdapter` set up for native API Level 11 fragments, in the `com.commonsware.cwac.pager` package.

PageDescriptor and PageEntry

`ArrayPagerAdapter` works with two representations of pages: `PageDescriptor` and `PageEntry`.

`PageDescriptor` is a simple interface, supplying the unique tag (`getFragmentTag()`) and indicator title (`getTitle()`) to use for a page:

```
package com.commonsware.cwac.pager;

import android.os.Parcelable;

public interface PageDescriptor extends Parcelable {
  String getFragmentTag();

  String getTitle();
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Developers can use `SimplePageDescriptor` as an implementation of `PageDescriptor` in most cases. `SimplePageDescriptor` just holds onto those two strings, plus handles the implementation of the `Parcelable` interface:

```
package com.commonsware.cwac.pager;

import android.os.Parcel;
import android.os.Parcelable;

public class SimplePageDescriptor implements PageDescriptor {
  private String tag=null;
  private String title=null;

  public static final Parcelable.Creator<SimplePageDescriptor> CREATOR=
      new Parcelable.Creator<SimplePageDescriptor>() {
        public SimplePageDescriptor createFromParcel(Parcel in) {
          return new SimplePageDescriptor(in);
        }

        public SimplePageDescriptor[] newArray(int size) {
          return new SimplePageDescriptor[size];
        }
      };

  public SimplePageDescriptor(String tag, String title) {
    this.tag=tag;
    this.title=title;
  }

  private SimplePageDescriptor(Parcel in) {
    tag=in.readString();
    title=in.readString();
  }

  @Override
  public int describeContents() {
    return(0);
  }

  @Override
  public void writeToParcel(Parcel out, int flags) {
    out.writeString(tag);
    out.writeString(title);
  }

  public String getTitle() {
    return(title);
  }

  public String getFragmentTag() {
    return(tag);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, the actual data model held by `ArrayPagerAdapter` is not the `PageDescriptor`, but rather a `PageEntry`, that holds onto its corresponding `PageDescriptor` *plus* a `Fragment.SavedState` object:

```
private static class PageEntry implements Parcelable {
  private PageDescriptor descriptor=null;
  private Fragment.SavedState state=null;

  public static final Parcelable.Creator<PageEntry> CREATOR=
      new Parcelable.Creator<PageEntry>() {
        public PageEntry createFromParcel(Parcel in) {
          return new PageEntry(in);
        }

        public PageEntry[] newArray(int size) {
          return new PageEntry[size];
        }
      };

  PageEntry(PageDescriptor descriptor) {
    this.descriptor=descriptor;
  }

  PageEntry(Parcel in) {
    this.descriptor=in.readParcelable(getClass().getClassLoader());
    this.state=in.readParcelable(getClass().getClassLoader());
  }

  PageDescriptor getDescriptor() {
    return(descriptor);
  }

  @Override
  public int describeContents() {
    return(0);
  }

  @Override
  public void writeToParcel(Parcel out, int flags) {
    out.writeParcelable(descriptor, 0);
    out.writeParcelable(state, 0);
  }
}
```

`Fragment.SavedState` is a `Parceble` object we can request from a `Fragment` at any point, representing the saved state of that fragment, as obtained via `onSaveInstanceState()` and related code. At present, that `Fragment.SavedState` is unused, as will be explained in the next section.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

RetentionStrategy

`ArrayPagerAdapter` also uses a `RetentionStrategy`, designed to abstract the logic for manipulating the fragments themselves as pages come and go within the `ViewPager`. `RetentionStrategy` is an interface, with methods to `attach()` a fragment to be in the pager and to `detach()` the fragment from the pager:

```
public interface RetentionStrategy {
  void attach(Fragment fragment, FragmentTransaction currTransaction);

  void detach(Fragment fragment, FragmentTransaction currTransaction);
```

There is only one stock implementation of this strategy at this time, in the form of a static data member named `KEEP`. This strategy is designed to replicate the behavior of `FragmentPagerAdapter`, keeping all fragments around once created, and merely `attach()`-ing and `detach()`-ing them from the `FragmentManager` as dictated:

```
  }
}

public static final RetentionStrategy KEEP=new RetentionStrategy() {
  @TargetApi(Build.VERSION_CODES.HONEYCOMB_MR2)
  public void attach(Fragment fragment,
                     FragmentTransaction currTransaction) {
    currTransaction.attach(fragment);
  }

  @TargetApi(Build.VERSION_CODES.HONEYCOMB_MR2)
  public void detach(Fragment fragment,
                     FragmentTransaction currTransaction) {
```

A future implementation of `ArrayPagerAdapter` should include another strategy that behaves more like `FragmentStatePagerAdapter`, removing the fragments entirely and allowing them to be garbage collected, while using `PageEntry` to hold onto their `Fragment.SavedState` structures to repopulate them later on if the user swipes back to that page.

Class Declaration and Generics

`ArrayPagerAdapter` uses Java generics to allow developers to state what `Fragment` subclass the pages are. This is for use with convenience methods — `getExistingFragment()` and `getCurrentFragment()` — to help reduce the developer's need to downcast those `Fragment` instances to some subclass. If the pages in the `ViewPager` will all come from a single `Fragment` subclass, the developer

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

would use that class as the `T` in the declaration; otherwise, the developer would just use `Fragment`:

```
abstract public class ArrayPagerAdapter<T extends Fragment> extends
```

Constructors

`ArrayPagerAdapter` offers two constructors. The simpler two-parameter constructor, taking the `FragmentManager` and the desired pages as an `ArrayList` of `PageDescriptor` objects, just chains to the three-parameter constructor. That third parameter is an instance of a `RetentionStrategy`, allowing reusers of `ArrayPagerAdapter` to try their own hand at implementing such a strategy. `null` — the default strategy from the standpoint of the constructors — is replaced with the default `KEEP` strategy, and the `PageDescriptor` objects are wrapped in `PageEntry` objects as the actual data model (an `entries` `ArrayList`):

```
  public ArrayPagerAdapter(FragmentManager fragmentManager,
                           List<PageDescriptor> descriptors,
                           RetentionStrategy retentionStrategy) {
    this.fm=fragmentManager;
    this.entries=new ArrayList<PageEntry>();

    for (PageDescriptor desc : descriptors) {
      validatePageDescriptor(desc);

      entries.add(new PageEntry(desc));
    }

    this.retentionStrategy=retentionStrategy;

    if (this.retentionStrategy == null) {
      this.retentionStrategy=KEEP;
    }
```

Core PagerAdapter Methods

All `PagerAdapter` implementations have some core methods that they must handle. When you create a subclass of `FragmentPagerAdapter` and `FragmentStatePagerAdapter`, you only need to worry about `getCount()` and `getPageTitle()`. However, if you are creating your own replacement for those fragment-based adapters, there are a few more standard `PagerAdapter` methods that you will need to override.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

getCount()

`getCount()` is easy: all we need to do is return our desired number of pages. That is based on the number of `PageDescriptor` objects supplied to our adapter, which we wrapped into `PageEntry` objects and hold onto in `entries`:

```
  @Override
  public int getCount() {
    return(entries.size());
```

getPageTitle()

Similarly, `getPageTitle()` just needs to find the appropriate `PageDescriptor` and call `getTitle()` on it, to supply the title for a given page for use by an indicator like `PagerTabStrip`:

```
    }
  }

  @Override
```

instantiateItem() and destroyItem()

The `instantiateItem()` method on `PagerAdapter` is responsible for setting up the user interface for a given page (indicated by `position`) and adding those widgets to a `ViewGroup` supplied as a parameter. It returns an `Object` that represents a "handle" to the page that `ViewPager` will return to the `PagerAdapter` in future calls, such as to `destroyItem()`.

A `Fragment`-based `PagerAdapter` can use the fragment itself as the "handle", and the fragment's `onCreateView()` as the means of obtaining the UI to pour into the `ViewGroup`.

Hence, the `ArrayPagerAdapter` implementation of `instantiateItem()` does the following:

- First, starts a `FragmentTransaction`, if there is not one already in progress
- Then, tries to find an existing `Fragment` for this `position`, using a `getExistingFragment()` helper method (described later in this chapter)
- If an existing fragment exists, `instantiateItem()` uses the `RetentionStrategy` to re-attach the UI

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- If an existing fragment does not exist, `instantiateItem()` calls the abstract `createFragment()` method, to allow the subclass to return the actual `Fragment` object given the `PageDescriptor`, then `add()` that fragment to the UI
- If the fragment is not already the current page, make sure that its action bar contributions are hidden via `setMenuVisibility()` and `setUserVisibleHint()`
- Return the fragment itself as the "handle"

```
@TargetApi(Build.VERSION_CODES.ICE_CREAM_SANDWICH_MR1)
@Override
public Object instantiateItem(ViewGroup container, int position) {
  if (currTransaction == null) {
    currTransaction=fm.beginTransaction();
  }

  Fragment fragment=getExistingFragment(position);

  if (fragment != null) {
    retentionStrategy.attach(fragment, currTransaction);
  }
  else {
    fragment=createFragment(entries.get(position).getDescriptor());
    currTransaction.add(container.getId(), fragment,
                        getFragmentTag(position));
  }

  if (fragment != currPrimaryItem) {
    fragment.setMenuVisibility(false);

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH_MR1) {
      fragment.setUserVisibleHint(false);
    }
  }

  return(fragment);
```

Conversely, `destroyItem()` is responsible for cleaning up anything from a page that the `PagerAdapter` thinks is no longer needed. The `destroyItem()` method on `ArrayPagerAdapter` starts a transaction if there is none, then delegates the actual work to the `RetentionStrategy`:

```
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
@Override
public void destroyItem(ViewGroup container, int position,
                        Object object) {
  if (currTransaction == null) {
    currTransaction=fm.beginTransaction();
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    retentionStrategy.detach((Fragment)object, currTransaction);
```

startUpdate() and finishUpdate()

The `startUpdate()` method will be called before any calls to `instantiateItem()` or `destroyItem()`, and so, if desired, we can do some initialization work there. In the case of `ArrayPagerAdapter`, all initialization is done lazily, and so `startUpdate()` is not needed. However, since `FragmentPagerAdapter` overrides `startUpdate()` with an empty implementation, we keep that for maximum fidelity with the stock implementation:

```
  @Override
  public void startUpdate(ViewGroup container) {
```

The `finishUpdate()` method will be called after any calls to `instantiateItem()` or `destroyItem()`, where we can do some cleanup work. `ArrayPagerAdapter` creates a `FragmentTransaction` as part of its work in `instantiateItem()` and `destroyItem()`, and so we need to commit that transaction in `finishUpdate()`. Once again, we reproduce the implementation from `FragmentPagerAdapter`, which uses `commitAllowingStateLoss()` (so we are not concerned with the timing of any state-saving being done at the activity level) and `executePendingTransactions()` (so all of the fragment work is done directly, rather than being posted to the end of the main application thread's work queue):

```
  @TargetApi(Build.VERSION_CODES.HONEYCOMB)
  @Override
  public void finishUpdate(ViewGroup container) {
    if (currTransaction != null) {
      currTransaction.commitAllowingStateLoss();
      currTransaction=null;
      fm.executePendingTransactions();
    }
```

setPrimaryItem()

`ViewPager` will call `setPrimaryItem()` on the `PagerAdapter` when a new page is being brought into view, based on gestures or other calls on `ViewPager` itself (e.g., `setCurrentItem()`). Some `PagerAdapter` implementations will have nothing much to do here. Fragment-based `PagerAdapter` implementations, though, need to ensure that the right fragment's action bar items are shown. Hence, `ArrayPagerAdapter` removes the action bar items from the previously-current page and adds the action bar items of the newly-current page:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @TargetApi(Build.VERSION_CODES.ICE_CREAM_SANDWICH_MR1)
  @SuppressWarnings("unchecked")
  @Override
  public void setPrimaryItem(ViewGroup container, int position,
                             Object object) {
    T fragment=(T)object;

    if (fragment != currPrimaryItem) {
      if (currPrimaryItem != null) {
        currPrimaryItem.setMenuVisibility(false);

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH_MR1)
{
          currPrimaryItem.setUserVisibleHint(false);
        }
      }

      if (fragment != null) {
        fragment.setMenuVisibility(true);

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH_MR1)
{
          fragment.setUserVisibleHint(true);
        }
      }

      currPrimaryItem=fragment;
    }
```

isViewFromObject()

`isViewFromObject()` helps `ViewPager` keep track of the UI for pages and how it maps back to a page's "handle". In our case, since the "handle" is a `Fragment`, we need to see if the supplied `View` is the `View` from the supplied `Fragment`:

```
  @TargetApi(Build.VERSION_CODES.HONEYCOMB)
  @Override
  public boolean isViewFromObject(View view, Object object) {
    return ((Fragment)object).getView() == view;
```

State Management

Our `PagerAdapter` is called with `saveState()` and `restoreState()` methods, to have us save the state of our data model and restore it, for configuration changes. `saveState()` returns a `Parcelable` which will form part of the state saved by the `ViewPager`, while `restoreState()` is handed back that `Parcelable` (or a copy).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The state of the *fragments* is handled by `FragmentManager`, no different than with any other fragments we might use in an activity. The mere fact that we happen to coordinate those fragments with a `PagerAdapter` does not change this. Hence, the "state" that we are dealing with in `saveState()` and `restoreState()` is solely the state of the `PagerAdapter` data model — in our case, the roster of pages.

To future-proof the implementation a bit, the state is represented as a `Bundle`, into which we can store other `Parcelable` objects. Since `Bundle` knows how to save an `ArrayList` of `Parcelable` objects, we can just call `putParcelableArrayList()` to save our `ArrayList` of `PageEntry` objects, restoring them in `restoreState()` via `getParcelableArrayList()`:

```
  @Override
  public Parcelable saveState() {
    Bundle state=new Bundle();

    state.putParcelableArrayList(KEY_DESCRIPTORS, entries);

    return(state);
```

Content Manipulation and Position Management

Perhaps the trickiest method on `PagerAdapter` that we have to worry about is innocuously named `getItemPostion()`. We are given the `Object` "handle" for a page, and we need to return the `position` of that page.

However, that's not really what is going on here.

`getItemPosition()` is used when we call `notifyDataSetChanged()` to indicate a structural change in our data model, such as an added or removed page. `ViewPager` is looking for `getItemPosition()` to tell us the *new* position of pages for this `notifyDataSetChanged()` call. So, as we manipulate our pages, we need to track what is going on with page positions, so `getItemPosition()` can return the correct data.

The actual value returned by `getItemPosition()` is either:

- The actual numerical position of the page, from `0` to `getCount()-1`, if the page moved to another position (where we return the new position)
- `PagerAdapter.POSITION_UNCHANGED`, if the page has not moved
- `PagerAdapter.POSITION_NONE`, if the page no longer exists (e.g., was removed)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

ArrayPagerAdapter simply holds a HashMap (positionDelta), mapping our Fragment "handle" to the page to an Integer representing any change to the position of that page made by methods like add(). When getItemPosition() is called, we return the value for that page out of the HashMap, or POSITION_UNCHANGED if the page does not appear in the HashMap, indicating that the page has not been affected:

```
    return(entries.get(position).getDescriptor().getTitle());
  }

  @Override
  public int getItemPosition(Object o) {
    Integer result=positionDelta.get(o);

    if (result == null) {
      return(PagerAdapter.POSITION_UNCHANGED);
    }
```

The add() method needs to add a new page to the data model, given the PageDescriptor. We clear() our positionDelta HashMap (as any previous changes should already have been picked up), add() a new PageEntry to our data model based on the supplied PageDescriptor, then call notifyDataSetChanged():

```
    return(result);
  }

  public void add(PageDescriptor desc) {
    validatePageDescriptor(desc);

    positionDelta.clear();
```

In this case, we did not need to add an entry to positionDelta, as ViewPager will use the natural position (based on where it appears in our data model) if we return POSITION_UNCHANGED.

The insert() method needs to do much the same thing, except rather than adding the new page to the end, we are adding it somewhere in the middle. This requires us to do everything we did in add(), plus add entries to the positionDelta map to indicate the new positions for every page that appears after the one being inserted:

```
    notifyDataSetChanged();
  }

  public void insert(PageDescriptor desc, int position) {
    validatePageDescriptor(desc);

    positionDelta.clear();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    for (int i=position; i < entries.size(); i++) {
      Fragment f=getExistingFragment(i);

      if (f != null) {
        positionDelta.put(f, i + 1);
      }
    }
```

The `remove()` method needs to get rid of an existing page, given its `position`. Here, we are not given the "handle", so we look it up via `getExistingFragment()`, then use that to put `POSITION_NONE` in the `positionDelta` map. We also update `positionDelta` to indicate the new positions for every page that appeared after the one being removed:

```
    notifyDataSetChanged();
  }

  public void remove(int position) {
    positionDelta.clear();

    Fragment f=getExistingFragment(position);

    if (f != null) {
      positionDelta.put(f, PagerAdapter.POSITION_NONE);
    }

    for (int i=position + 1; i < entries.size(); i++) {
      f=getExistingFragment(i);

      if (f != null) {
        positionDelta.put(f, i - 1);
      }
    }
```

Finally, a `move()` is simply treated as a `remove()` from the old position and an `insert()` of the same page into the new position:

```
    notifyDataSetChanged();
  }

  public void move(int oldPosition, int newPosition) {
    if (oldPosition != newPosition) {
      PageDescriptor desc=entries.get(oldPosition).getDescriptor();

      remove(oldPosition);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Miscellany

One headache with `FragmentPagerAdapter` and `FragmentStatePagerAdapter` is that they like to manage the fragments themselves, making it annoying for you to get at those fragments independently later on. Some developers have resorted to holding onto fragments in their own array, which works, but then you run into problems when it comes to garbage collection with `FragmentStatePagerAdapter`.

`ArrayPagerAdapter` provides two convenience methods to address this:

- `getExistingFragment()` simply returns the fragment for a given `position`, by finding the tag for that fragment from the `PageDescription`, then looking up the fragment by that tag. This way, if the fragment does not exist due to garbage collection, we can return `null`
- `getCurrentFragment()` returns the `currPrimaryItem` value, indicating the page that we are presently on

```
    return (T)(fm.findFragmentByTag(getFragmentTag(position)));
  }
```

Both are set to return the generic type `T` that the developer uses when creating a subclass of `ArrayPagerAdapter`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Focus Management and Accessibility

As developers, we are very used to creating apps that are designed to be navigated by touch, with users tapping on widgets and related windows to supply input.

However, not all Android devices have touchscreens, and not all Android users use touchscreens.

Internationalization (i18n) and localization (L10n) give you opportunities to expand your user base to audiences beyond your initial set, based on language. Similarly, you can expand your user base by offering support for non-touchscreen input and output. Long-term, the largest user base of these features may be those with televisions augmented by Android, whether via Android TV, OUYA consoles, or whatever. Short-term, the largest user base of these features may be those for whom touchscreens are rarely a great option, such as the blind. Supporting those with unusual requirements for input and output is called *accessibility* (a11y), and represents a powerful way for you to help your app distinguish itself from competitors.

In this chapter, we will first examine how to better handle focus management, and then segue into examining what else, beyond supporting keyboard-based input, can be done in the area of accessibility.

Prerequisites

Understanding this chapter requires that you have read the core chapters and are familiar with the concept of widgets having focus for user input.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Prepping for Testing

To test focus management, you will need an environment that supports "arrow key" navigation. Here, "arrow key" also includes things like D-pads or trackballs – basically, anything that navigates by key events instead of by touch events.

Examples include:

- The Android emulator, with the `DPad support` hardware property set to `yes`
- Phones that have actual D-pads, trackballs, arrow keys, or the like
- Television-based Android environments, such as Android TV or the OUYA console
- Devices that have dedicated keyboard accessories, such as the keyboard "slice" available for the ASUS Transfomer series of tablets
- A standard Android device accessed via a Bluetooth keyboard, gamepad, or similar sort of pointing device

Hence, even if the emulator will be insufficient for your needs, you should be able to set up a hardware test environment relatively inexpensively. Most modern Android devices support Bluetooth keyboards, and such keyboards frequently can be obtained at low relative cost.

For accessibility beyond merely focus control, you will certainly want to enable TalkBack, via the Accessibility area of the Settings app. This will cause Android to verbally announce what is on the screen, by means of its text-to-speech engine.

On Android 4.0 and higher devices, enabling Talkback will also optionally enable "Explore by Touch". This allows users to tap on items (e.g., icons in a `GridView`) to have them read aloud via TalkBack, with a double-tap to actually perform what ordinarily would require a single-tap without "Explore by Touch".

Controlling the Focus

Android tries its best to have intelligent focus management "out of the box", without developer involvement. Many times, what it offers is sufficient for your needs. Other times, though, the decisions Android makes are inappropriate:

- Trying to navigate in a certain direction (e.g., right) moves focus to a widget that is not logically what should have the focus

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Focus has other side effects, like showing the soft keyboard on an `EditText` widget, that is not desirable

Hence, if you feel that you need to take more control over how focus management is handled, you have many means of doing so, covered in this section.

Establishing Focus

In order for a widget to get the focus, it has to be focusable.

You might think that the above sentence was just a chance for the author to be witty. It was... a bit. But there are actually two types of "focusable" when it comes to Android apps:

- Is it focusable when somebody is using a pointing device or the keyboard?
- Is it focusable in touch mode?

There are three major patterns for the default state of a widget:

1. Some are initially focusable in both cases (e.g., `EditText`)
2. Some are focusable in non-touch mode but are not focusable in touch mode (e.g., `Button`)
3. Some are not focusable in either mode (e.g., `TextView`)

So, when a `Button` is not focusable in touch mode, that means that while the button will take the focus when the user navigates to it (e.g., via keys), the button will *not* take the focus when the user simply taps on it.

You can control the focus semantics of a given widget in four ways:

- You can use `android:focusable` and `android:focusableInTouchMode` in a layout
- You can use `setFocusable()` and `setFocusableInTouchMode()` in Java

We will see examples of these shortly.

Requesting (or Abandoning) Focus

By default, the focus will be granted to the first focusable widget in the activity, starting from the upper left. Often times, this is a fine solution.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you want to have some other widget get the focus (assuming that the widget is focusable, per the section above), you have two choices:

1. Call `requestFocus()` on the widget in question
2. You can give the widget's layout element a child element, named `<requestFocus />`, to stipulate that this widget should be the one to get the focus

Note that this is a child element, not an attribute, as you might ordinarily expect.

For example, let's look at the Focus/Sampler sample project, which we will use to illustrate various focus-related topics.

Our main activity, creatively named `MainActivity`, loads a layout named `request_focus.xml`, and demonstrates the `<requestFocus />` element:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/a_button"/>

  <EditText
    android:id="@+id/editText1"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:contentDescription="@string/first_field"
    android:hint="@string/str_1st_field"
    android:inputType="text"/>

  <EditText
    android:id="@+id/editText2"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:contentDescription="@string/second_field"
    android:hint="@string/str_2nd_field"
    android:inputType="text">

    <requestFocus/>
  </EditText>

</LinearLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here, we have three widgets in a horizontal `LinearLayout`: a `Button`, and two `EditText` widgets. The second `EditText` widget has the `<requestFocus />` child element, and so it gets the focus when we display our launcher activity:

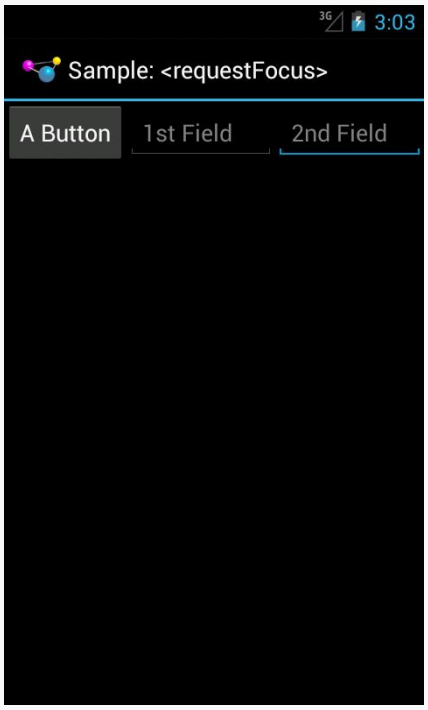


Figure 550: Focus Sampler, Showing Requested Focus

If we had skipped the `<requestFocus />` element, the focus would have wound up on the first `EditText`... assuming that we are working in touch mode. If the activity had been launched via the pointing device or keyboard, then the `Button` would have the focus, because the `Button` is focusable in non-touch mode by default.

Calling `requestFocus()` from Java code gets a bit trickier. There are a few flavors of the `requestFocus()` method on `View`, of which two will be the most popular:

- An ordinary zero-argument `requestFocus()`
- A one-argument `requestFocus()`, with the argument being the direction in which the focus should theoretically be coming from

You might look at the description of the second flavor and decide that the zero-argument `requestFocus()` looks a lot easier. And, sometimes it will work. However, sometimes it will not, as is the case with our second activity, `RequestFocusActivity`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In this activity, our layout (`focusable_button`) is a bit different:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <EditText
    android:id="@+id/editText1"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:contentDescription="@string/first_field"
    android:hint="@string/str_1st_field"
    android:inputType="text"/>

  <EditText
    android:id="@+id/editText2"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:contentDescription="@string/second_field"
    android:hint="@string/str_2nd_field"
    android:inputType="text">
  </EditText>

  <Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:focusableInTouchMode="true"
    android:text="@string/a_button"/>

</LinearLayout>
```

Here, we put the `Button` last instead of first. We have no `<requestFocus />` element anywhere, which would put the default focus on the first `EditText` widget. And, our `Button` has `android:focusableInTouchMode="true"`, so it will be focusable regardless of whether we are in touch mode or not.

In `onCreate()` of our activity, we use the one-parameter version of `requestFocus()` to give the `Button` the focus:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.focusable_button);
    initActionBar();

    button=findViewById(R.id.button1);
    button.requestFocus(View.FOCUS_RIGHT);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    button.setOnClickListener(this);
  }
```

If there were only the one `EditText` before the `Button`, the zero-argument `requestFocus()` works. However, with a widget between the default focus and our `Button`, the zero-argument `requestFocus()` does not work, but using `requestFocus(View.FOCUS_RIGHT)` does. This tells Android that we want the focus, and it should be as if the user is moving to the right from where the focus currently lies.

All of our activities inherit from a `BaseActivity` that manages our action bar, with an overflow menu to get to the samples and the app icon to get to the original activity.

So, if you run the app and choose "Request Focus" from the overflow menu, you will see:

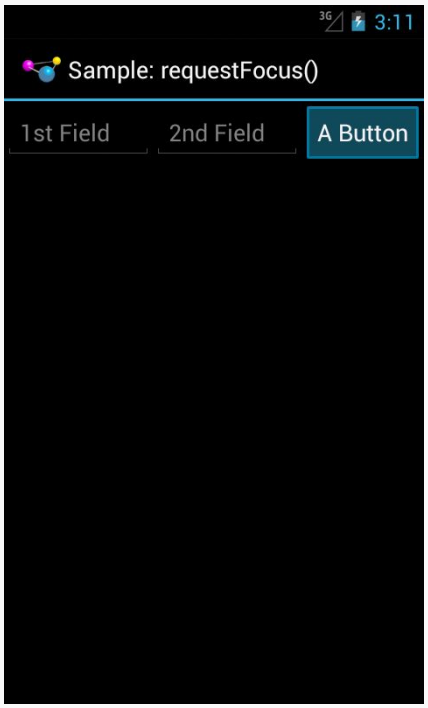


Figure 551: Focus Sampler, Showing Manually-Requested Focus

We also wire up the `Button` to the activity for click events, and in `onClick()`, we call `clearFocus()` to abandon the focus:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@Override
public void onClick(View v) {
  button.clearFocus();
}
```

What `clearFocus()` will do is return to the original default focus for this activity, in our case the first `EditText`:

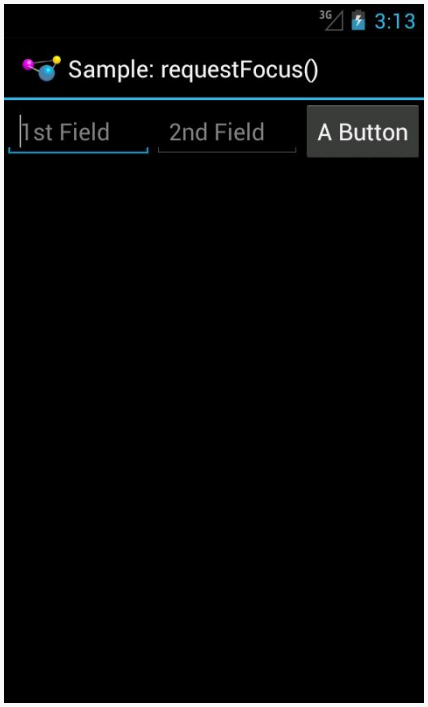


Figure 552: Focus Sampler, After Clearing the Focus

Focus Ordering

Beyond manually placing the focus on a widget (or manually clearing that focus), you can also override the focus order that Android determines automatically. While Android's decisions usually are OK, they may not be optimal.

A widget can use `android:nextFocus...` attributes in the layout file to indicate the widget that should get control on a focus change in the direction indicated by the `...` part. So, `android:nextFocusDown`, applied to Widget A, indicates which widget should receive the focus if, when the focus is on Widget A, the user "moves down" (e.g., presses a DOWN key, presses the down direction on a D-pad). The same logic holds true for the other three directions (`android:nextFocusLeft`, `android:nextFocusRight`, and `android:nextFocusUp`).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For example, the `res/layout/table.xml` resource in the `FocusSampler` project is based on the `TableLayout` sample from early in this book, with a bit more focus control:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:stretchColumns="1">

  <TableRow>

    <TextView android:text="@string/url"/>

    <EditText
      android:id="@+id/entry"
      android:layout_span="3"
      android:inputType="text"
      android:nextFocusRight="@+id/ok"/>
  </TableRow>

  <TableRow>

    <Button
      android:id="@+id/cancel"
      android:layout_column="2"
      android:text="@string/cancel"/>

    <Button
      android:id="@+id/ok"
      android:text="@string/ok"/>
  </TableRow>

</TableLayout>
```

In the original `TableLayout` sample, by default, pressing either RIGHT or DOWN while the `EditText` has the focus will move the focus to the "Cancel" button. This certainly works. However, it does mean that there is no single-key means of moving from the `EditText` to the "OK" button, and it would be nice to offer that, so those using the pointing device or keyboard can quickly move to either button.

This is a matter of overriding the default focus-change behavior of the `EditText` widget. In our case, we use `android:nextFocusRight="@+id/ok"` to indicate that the "OK" button should get the focus if the user presses RIGHT from the `EditText`. This gives RIGHT and DOWN different behavior, to reach both buttons.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Scrolling and Focusing Do Not Mix

Let's suppose that you have a UI design with a fixed bar of widgets at the top (e.g., action bar), a `ListView` dominating the activity, and a panel of widgets at the bottom (e.g., button panel, or a split action bar), such as this book's original action bar demo:

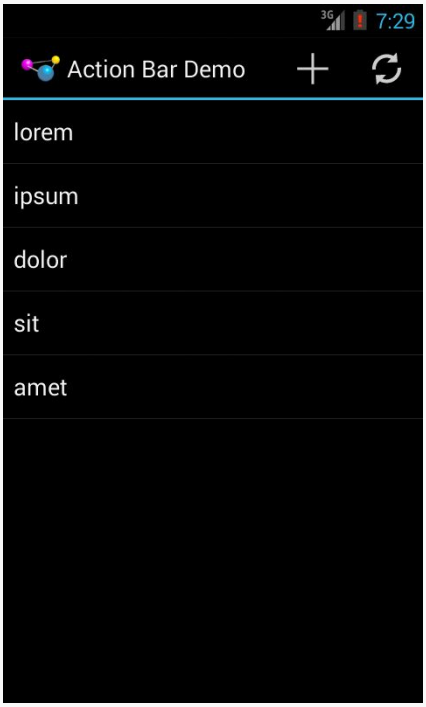


Figure 553: A Split Action Bar

This is a common UI pattern, much to the detriment of those using pointing devices or keyboards for navigation. In order to get to the bottom panel of widgets, they will have to scroll through the entire list first, because scrolling trumps focus changes. So while this is easy to navigate via a touchscreen, it is a major problem to navigate for those not using a touchscreen.

Similarly, if the user has scrolled down the list, and now wishes to get to the top action bar, the user would have to scroll all the way to the top of the list first.

Workarounds include:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Overriding focus control such that left and right navigation from the list moves you to the action bar or button panel (e.g., left moves you to the action bar, right moves you to the button panel)
- In a television setup, having the "action bar" be vertical down the left, and the button panel be vertical down the right, so you automatically get the left/right navigation to move between these "zones"
- Eliminating the button panel, moving those items instead to the action bar, or perhaps an action mode (a.k.a., contextual action bar) if the buttons are only relevant if the user checks one or more items in the list
- Offer a hotkey, separate from navigation, that repositions the focus (e.g., CTRL-A to jump to the action bar), if you believe that users will read your documentation to discover this key combination

In the specific case of a split action bar, Android handles this for you: moving focus to the right from the top action bar moves you to the bottom action bar directly, whereas moving focus down from the top action bar moves you into your main content view (e.g., the `ListView` in the action bar sample image shown above).

Accessibility and Focus

People suffering from impaired vision, including the blind, have had to rely heavily on proper keyboard navigation for their use of Android apps, at least prior to Android 4.0 and "Explore by Touch". These users need focus to be sensible, so that they can find their way through your app, with TalkBack supplying prompts for what has the focus. Having widgets that are unreachable in practice will eliminate features from your app for this audience, simply because they cannot get to them.

"Explore by Touch" provides accessibility assistance without reliance upon proper focus. However:

- "Explore by Touch" is new to Android 4.0, and many visually-impaired users will be using older devices, particularly through 2013
- "Explore by Touch" is less reliable than keyboard-based navigation, insofar as users have to remember specific screen locations (and get to them without seeing those locations), rather than simply memorizing certain key combinations
- "Explore by Touch", by requiring additional taps (e.g., double-tap to tap a `Button`), may cause some challenges when the UI itself requires additional taps (e.g., a double-tap on a widget to perform an action — is this now a triple-tap in "Explore by Touch" mode?)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- "Explore by Touch" is mostly for the visually impaired, and does not help others that might benefit from key-based navigation (e.g., people with limited motor control)

So, even though "Explore by Touch" will help people use apps that cannot be navigated purely through key events, the better you can support keyboards, the better off your users will be.

Accessibility Beyond Focus

While getting focus management correct goes a long way towards making your application easier to use, it is not the only thing to consider for making your application truly accessible by all possible users. This section covers a number of other things that you should consider as part of your accessibility initiatives.

Content Descriptions

For TalkBack to work, it needs to have something useful to read aloud to the user. By default, for most widgets, all it can say is the type of widget that has the focus (e.g., "a checkbox"). That does not help the TalkBack-reliant user very much.

Please consider adding `android:contentDescription` attributes to most of your widgets, pointing to a string resource that briefly describes the widget (e.g., "the Enabled checkbox"). This will be used in place of the basic type of widget by TalkBack.

Classes that inherit from `TextView` will use the text caption of the widget by default, so your `Button` widgets may not need `android:contentDescription` if their captions will make sense to TalkBack users.

However, with an `EditText`, since the text will be what the user types in, the text is not indicative of the widget itself. Android will first use your `android:hint` value, if available, falling back to `android:contentDescription` if `android:hint` is not supplied.

Also, bear in mind that if the widget changes purpose, you need to change your `android:contentDescription` to match. For example, suppose you have a media player app with an `ImageButton` that you toggle between "play" and "pause" modes by changing its image. When you change the image, you also need to change the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`android:contentDescription` as well, lest sighted users think the button will now "pause" while blind users think that the button will now "play".

Custom Widgets and Accessibility Events

The engine behind TalkBack is an accessibility service. Android ships with some, like TalkBack, and third parties can create other such services.

Stock Android widgets generate relevant accessibility events to feed data into these accessibility services. That is how `android:contentDescription` gets used, for example — on a focus change, stock Android widgets will announce the widget that just received the focus.

If you are creating custom widgets, you may need to raise your own accessibility events. This is particularly true for custom widgets that draw to the `Canvas` and process raw touch events (rather than custom widgets that merely aggregate existing widgets).

The Android developer documentation provides instructions for when and how to supply these sorts of events.

Announcing Events

Sometimes, your app will change something about its visual state in ways that do not get picked up very well by any traditional accessibility events. For example, you might use `GestureDetector` to handle some defined library of gestures and change state in your app. Those state changes may have visual impacts, but `GestureDetector` will not know what those are and therefore cannot supply any sort of accessibility event about them.

To help with this, API Level 16 added `announceForAccessibility()` as a method on `View`. Just pass it a string and that will be sent out as an "announcement" style of `AccessibilityEvent`. Your code leveraging `GestureDetector`, for example, could use this to explain the results of having applied the gesture.

Font Selection and Size

For users with limited vision, being able to change the font size is a big benefit. Android 4.0 finally allows this, via the Settings app, so users can choose between

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

small, normal, large, and huge font sizes. Any place where text is rendered and is measured in `sp` will adapt.

The key, of course, is the `sp` part.

`sp` is perhaps the most confusing of the available dimension units in Android. `px` is obvious, and `dp` (or `dip`) is understandable once you recognize the impacts of screen density. Similarly, `in`, `mm`, and `pt` are fairly simple, at least once you remember that `pt` is 1/72nd of an inch.

If the user has the font scale set to "normal", `sp` equates to `dp`, so a dimension of `30sp` and `30dp` will be the same size. However, values in `dp` do not change based on font scale; values in `sp` will increase or decrease in physical size based upon the user's changes to the font scale.

We can see how this works in the Accessibility/FontScale sample project.

In our layout (`res/layout/activity_main.xml`), we have six pieces of text: two each (regular and bold) measured at `30px`, `30dp`, and `30sp`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/LinearLayout1"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
    android:text="@string/normal_30px"
    android:textSize="30px"
    tools:context=".MainActivity"/>

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/bold_30px"
    android:textSize="30px"
    android:textStyle="bold"
    tools:context=".MainActivity"/>

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
    android:text="@string/normal_30dp"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:textSize="30dp"
    tools:context=".MainActivity"/>

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/bold_30dp"
    android:textSize="30dp"
    android:textStyle="bold"
    tools:context=".MainActivity"/>

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
    android:text="@string/normal_30sp"
    android:textSize="30sp"
    tools:context=".MainActivity"/>

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/bold_30sp"
    android:textSize="30sp"
    android:textStyle="bold"
    tools:context=".MainActivity"/>

</LinearLayout>
```

You will be able to see the differences between `30px` and `30dp` on any Android OS release, simply by running the app on devices with different densities. To see the changes between `30dp` and `30sp`, you will need to run the app on an Android 4.0+ device or emulator and change the font scale from the Settings app (typically in the Display section).

Here is what the text looks like with a normal font scale:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

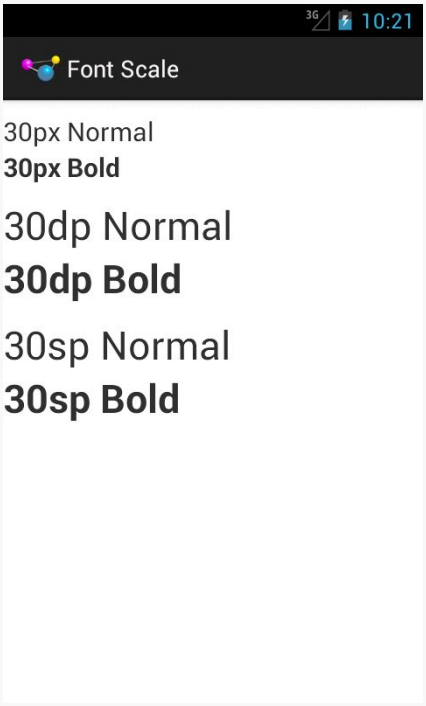


Figure 554: Fonts at Normal Scale

As you can see, `30dp` and `30sp` are equivalent.

If we raise the font scale to "large", the `30sp` text grows to match:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

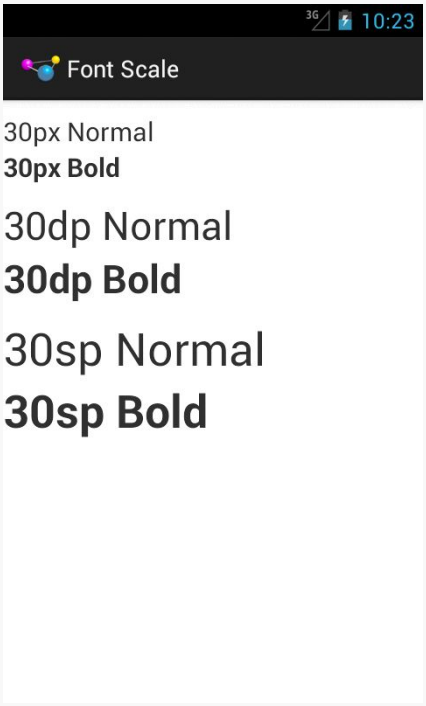


Figure 555: Fonts at Large Scale

Moving to "huge" scale increases the `30sp` text size further:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

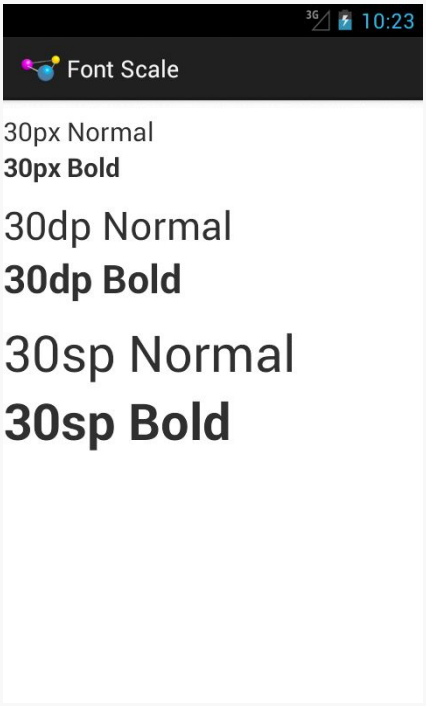


Figure 556: Fonts at Huge Scale

In the other direction, some users may elect to drop their font size to "small", with a corresponding impact on the `30sp` text:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

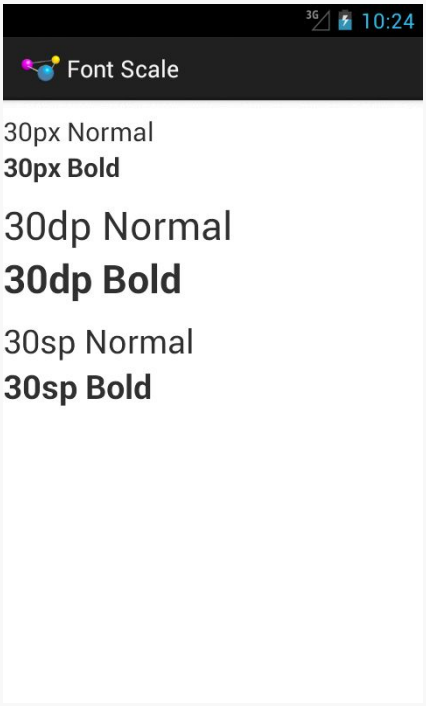


Figure 557: Fonts at Small Scale

As a developer, your initial reaction may be to run away from `sp`, because you do not control it. However, just as Web developers should deal with changing font scale in Web browsers, Android developers should deal with changing font scale in Android apps. Remember: the user is changing the font scale because the *user* feels that the revised scale is easier for them to use. Blocking such changes in your app, by avoiding `sp`, will not be met with love and adoration from your user base.

Also, bear in mind that changes to the font scale represent a configuration change. If your app is in memory at the time the user goes into Settings and changes the scale, if the user returns to your app, each activity that comes to the foreground will undergo the configuration change, just as if the user had rotated the screen or put the device into a car dock or something.

Widget Size

Users with ordinary sight already have trouble with tiny widgets, as they are difficult to tap upon.

Users trying to use the Explore by Touch facility added in Android 4.1 have it worse, as they cannot even see (or see well) the tiny target you are expecting them to tap

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

upon. They need to be able to reliably find your widget based on its relative position on the screen, and their ability to do so will be tied, in part, on widget size.

The Android design guidelines recommend 7–10mm per side minimum sizes for tappable widgets. In particular, they recommend 48dp per side, which results in a size of about 9mm per side.

You also need to consider how closely packed your widgets are. The closer the tap targets lie, the more likely it is that all users — whether using Explore by Touch or not — will accidentally tap on the wrong thing. Google recommends 8dp or more of margin between widgets. Also note that the key is *margins*, as while increasing padding might visually separate the widgets, the padding is included as part of the widget from the standpoint of touch events. While padding may help users with ordinary sight, margins provide similar help while also being of better benefit to those using Explore by Touch.

Gestures and Taps

If you employ gestures, be careful when employing the same gesture in different spots for different roles, particularly within the same activity.

For example, you might use a horizontal swipe to the right to switch pages in a `ViewPager` in some places and remove items from a `ListView` in others. While there may be visual cues to help explain this to users with ordinary sight, it may be far less obvious what is going on for TalkBack users. This is even more true if you are somehow combining these things (e.g., the `ListView` in question is in a page of the `ViewPager`).

Also, be a bit careful as you "go outside the box" for tap events. You might decide that a double-tap, or a two-finger tap, has special meaning on some widgets. Make sure that this still works when users use Explore by Touch, considering that the first tap will be "consumed" by Explore by Touch to announce the widget being tapped upon.

Enhanced Keyboard Support

All else being equal, users seeking accessibility assistance will tend to use keyboards when available. For users with limited (or no) sight, tactile keyboards are simply easier to use than touchscreens. For users with limited motor control, external devices that interface as keyboards may allow them to use devices that otherwise they could not.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Of course, plenty of users will use keyboards outside of accessibility as well. For example, devices like the ASUS Transformer series form perfectly good "netbook"-style devices when paired with their keyboards.

Hence, consider adding hotkey support, to assist in the navigation of your app. Some hotkeys may be automatically handled (e.g., Ctrl-C for copy in an `EditText`). However, in other cases you may wish to add those yourself (e.g., Ctrl-C for "copy" with respect to a checklist and its selected rows, in addition to a "copy" action mode item).

API Level 11 adds `KeyEvent` support for methods like `isCtrlPressed()` to detect meta keys used in combination with regular keys.

Audio and Haptics

Of course, another way to make your app more accessible is to provide alternative modes of input and output, beyond the visual.

Audio is popular in this regard:

- Using tones or clicks to reinforce input choices
- Integrating your own text-to-speech to augment TalkBack
- Integrating speech recognition for simple commands

However, bear in mind that deaf users will be unable to hear your audio. You are better served using both auditory and visual output, not just one or the other.

In some cases, haptics can be helpful for input feedback, by using the `Vibrator` system service to power the vibration motor. While most users will be able to feel vibrations, the limitation here is whether the device is capable of vibrating:

- Some tablets lack a vibration motor
- Television-based Android environment may or may not have some sort of vibration output (e.g., remote controls probably will not, but game controllers might)
- Devices not held in one's hand, such as those in a dock, will make haptics less noticeable

So, audio and vibration can help augment visual input and output, though they should not be considered complete replacements except in rare occurrences.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Color and Color Blindness

Approximately 8% of men (and 0.5% of women) in the world are colorblind, meaning that they cannot distinguish certain close colors:

> ...It's not that colorblind people (in most cases) are incapable or perceiving "green," instead they merely distinguish fewer shades of green than you do. So where you see three similar shades of green, a colorblind user might only see one shade of green.

(from "Tips for Designing for Colorblind Users")

Hence, relying solely on colors to distinguish different items, particularly when required for user input, is not a wise move.

Make sure that there is something more to distinguish two pieces of your UI than purely a shift in color, such as:

- Labels or icons
- Textures (e.g., solid vs. striped)
- Borders (e.g., drop shadow)

Accessibility Beyond Impairment

Accessibility is often tied to impaired users: ones with limited (or no) sight, ones with limited (or no) hearing, ones with limited motor control, etc.

In reality, accessibility is for *situations* where users may have limitations. For example, a user who might not normally think of himself as "impaired" has limited sight, hearing, and motor control when those facilities are already in use, such as while driving.

Hence, offering features that help with accessibility can benefit *all* your users, not just ones you think of as "impaired". For example:

- Offer a UI mode with an eye towards use in low-visibility situations that can either be manually invoked (e.g., via a preference) or automatically invoked (e.g., via a car dock)
- Offer voice input (commands) and output (text-to-speech) — iOS's Siri is not just for the blind, after all

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Offer hotkeys, not only to help those requiring a keyboard as their primary mode of input (e.g., blind users minimizing touchscreen use), but to help those who opt into using it for input (e.g., using a keyboard with an Android tablet in lieu of a traditional notebook or netbook)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Miscellaneous UI Tricks

While well-written GUI frameworks are better organized than XKCD's take on home organization, there are always a handful of tidbits that do, indeed, get categorized as "miscellaneous".

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Having an appreciation for XKCD is welcome, but optional.

Full-Screen and Lights-Out Modes

Full-screen mode, in Android parlance, means removing any system-supplied "bars" from the screen: title bar, action bar, status bar, system bar, navigation bar, etc. You might use this for games, video players, digital book readers, or other places where the time spent in an activity is great enough to warrant removing some of the normal accouterments to free up space for whatever the activity itself is doing.

Lights-out mode, in Android parlance, is where you take the system bar or navigation bar and dim the widgets inside of them, such that the bar is still usable, but is less visually distracting. This is a new concept added in Android 3.0 and has no direct analogue in Android 1.x or 2.x.

Android 1.x/2.x

To have an activity be in full-screen mode, you have two choices:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. Having the activity use a theme of `Theme.NoTitleBar.Fullscreen` (or some custom theme that inherits from `Theme.NoTitleBar.Fullscreen`)
2. Execute the following statements in `onCreate()` of your activity *before* calling `setContentView()`:

```
requestWindowFeature(Window.FEATURE_NO_TITLE);
getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                     WindowManager.LayoutParams.FLAG_FULLSCREEN);
```

The first statement removes the title bar or action bar. The second statement indicates that you want the activity to run in full-screen mode, hiding the status bar.

Android 4.0+

Things got significantly more messy once we started adding in the system bar (and, later, the navigation bar as the replacement for the system bar). Since these bars provide the user access to HOME, BACK, etc., it is usually important for them to be available. Android's behavior, therefore, varies in how you ask for something to happen and what then happens, based upon whether the device is a phone or a tablet.

The Activities/FullScreen sample project tries to enumerate some of the possibilities. On an Android 4.0 device, we have three `RadioButtons`:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <RadioGroup
    android:id="@+id/screenStyle"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <RadioButton
      android:id="@+id/normal"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:checked="true"
      android:text="@string/display_normal"/>

    <RadioButton
      android:id="@+id/lowProfile"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/display_low_profile"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    <RadioButton
      android:id="@+id/hideNav"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/display_hide_navigation"/>
  </RadioGroup>

  <Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:text="@string/something_at_the_bottom"/>

</RelativeLayout>
```

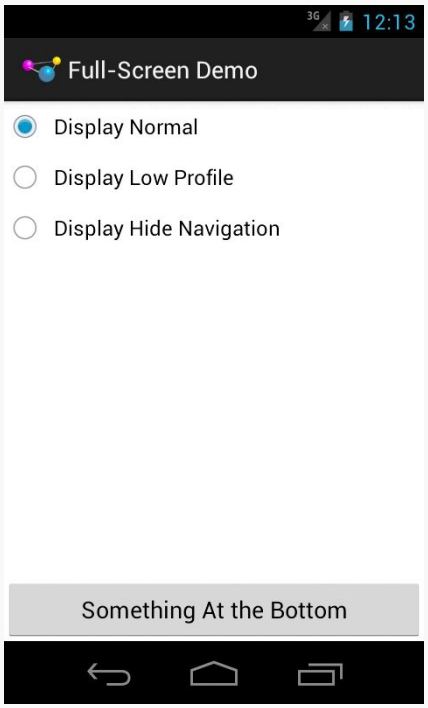


Figure 558: Sample UI, As Initially Launched on Android 4.0

...while on Android 4.1 or higher, we have another two possibilities:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <RadioGroup
    android:id="@+id/screenStyle"
    android:layout_width="match_parent"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:layout_height="wrap_content">

    <RadioButton
      android:id="@+id/normal"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:checked="true"
      android:text="@string/display_normal"/>

    <RadioButton
      android:id="@+id/lowProfile"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/display_low_profile"/>

    <RadioButton
      android:id="@+id/hideNav"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/display_hide_navigation"/>

    <RadioButton
      android:id="@+id/hideStatusBar"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/hide_status_bar"/>

    <RadioButton
      android:id="@+id/fullScreen"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/display_full_screen"/>
  </RadioGroup>

  <Button
    android:id="@+id/button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:text="@string/something_at_the_bottom"/>

</RelativeLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

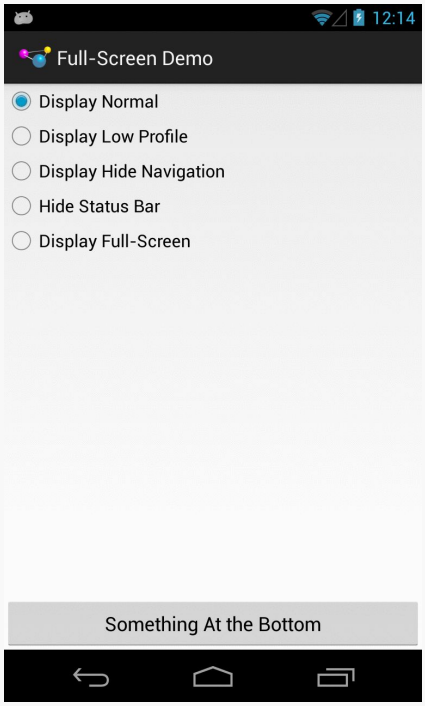


Figure 559: Sample UI, As Initially Launched on a Nexus 4/Android 4.2

Controlling the full-screen and lights-out modes is managed via a call to `setSystemUiVisibility()`, a method on `View`. You pass in a value made up of an OR'd (`|`) set of flags indicating what you want the visibility to be, with the default being normal operation. Hence, in the screenshot above, you see a Nexus 4 in normal mode. Here is the same UI on a Nexus 10 in normal mode:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

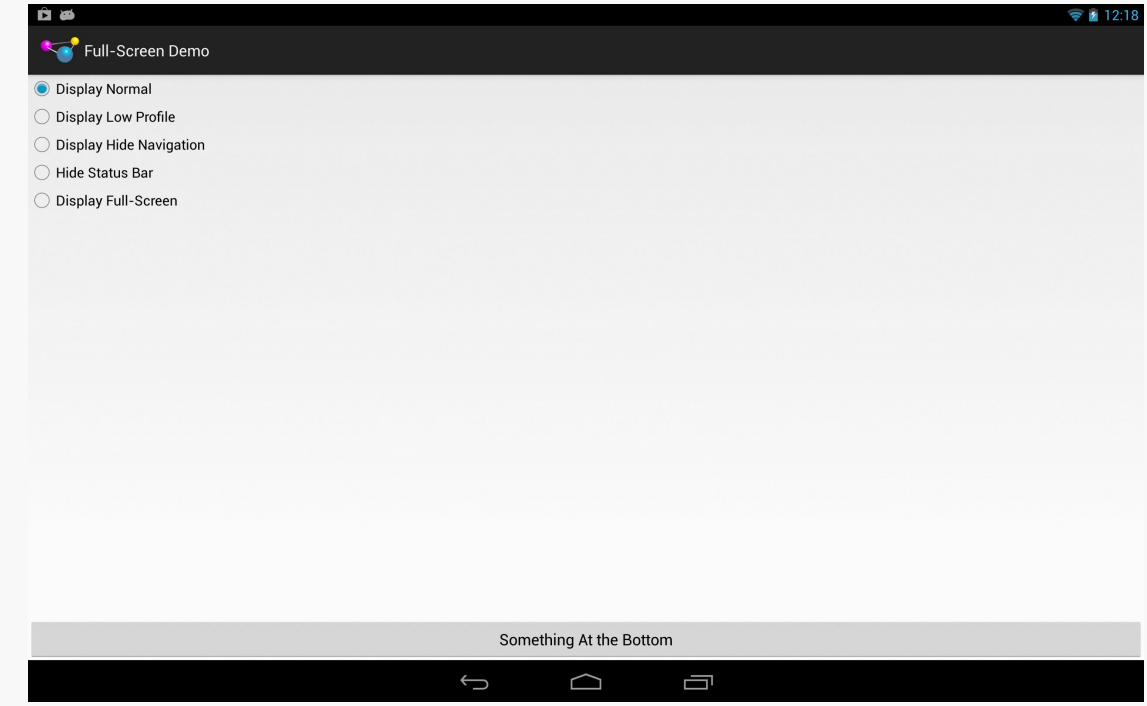


Figure 560: Sample UI, As Initially Launched on a Nexus 10/Android 4.2

Lights-out, or low-profile mode, is achieved by calling `setSystemUiVisibility()` with the `View.SYSTEM_UI_FLAG_LOW_PROFILE` flag. This will dim the navigation or system bar, so the bar is there and the buttons are still active, but that they are less visually intrusive:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

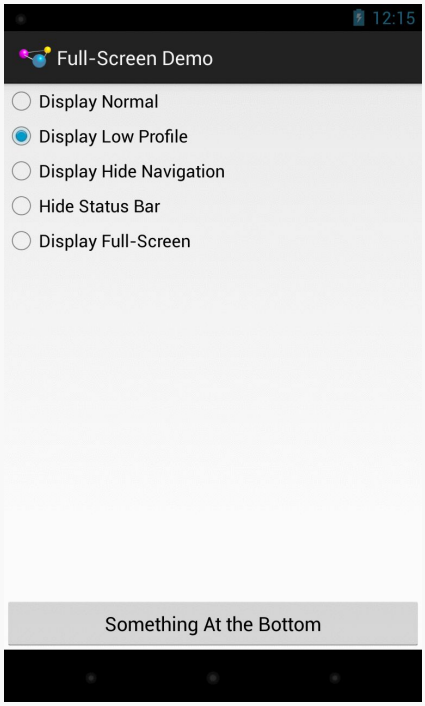


Figure 561: Sample UI, Lights-Out Mode, Nexus 4/Android 4.2

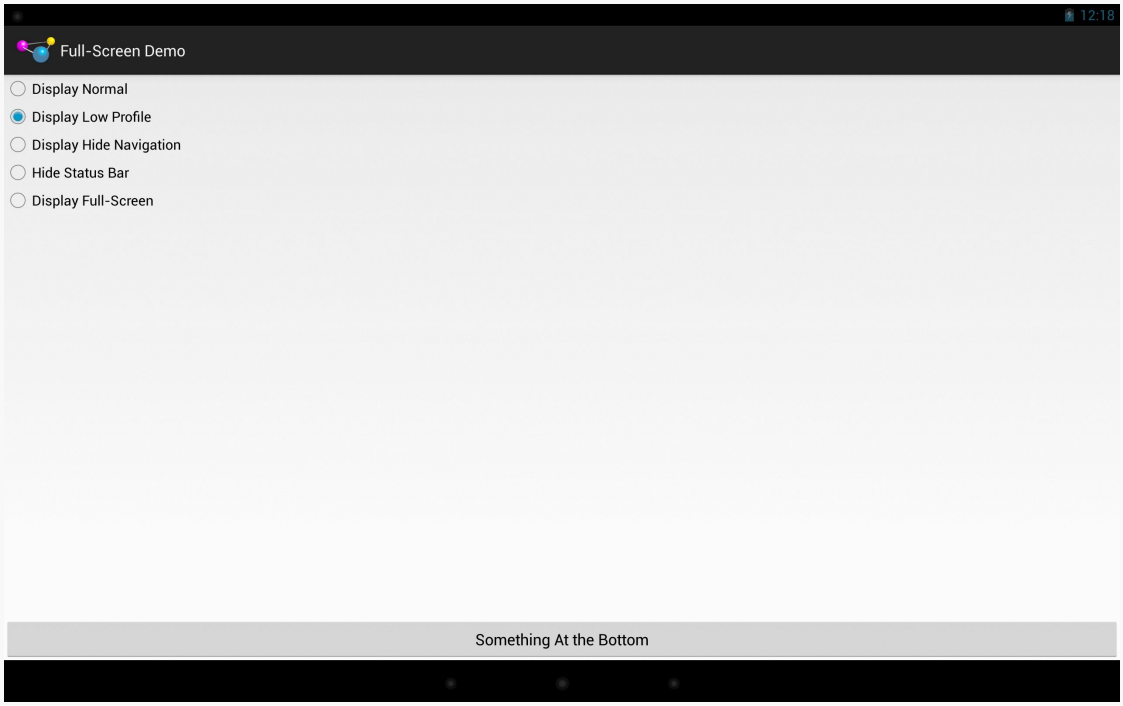


Figure 562: Sample UI, Lights-Out Mode, Nexus 10/Android 4.2

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You can temporarily hide the navigation bar (or system bar) by passing `View.SYSTEM_UI_FLAG_HIDE_NAVIGATION` to `setSystemUiVisibility()`. The bar will disappear, until the user touches the UI, in which case the bar reappears:

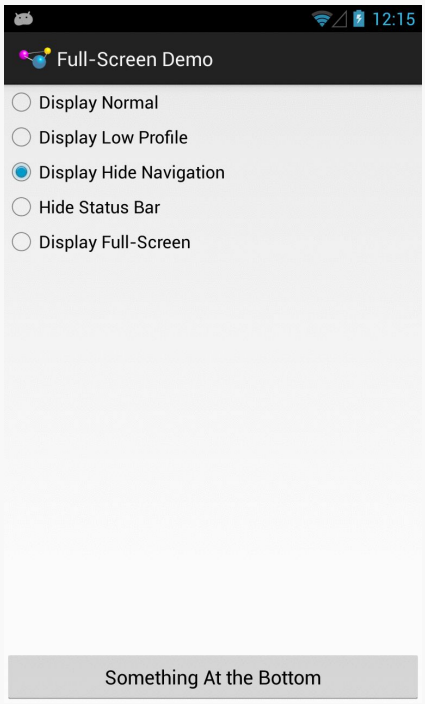


Figure 563: Sample UI, Hidden-Navigation Mode, Nexus 4/Android 4.2

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

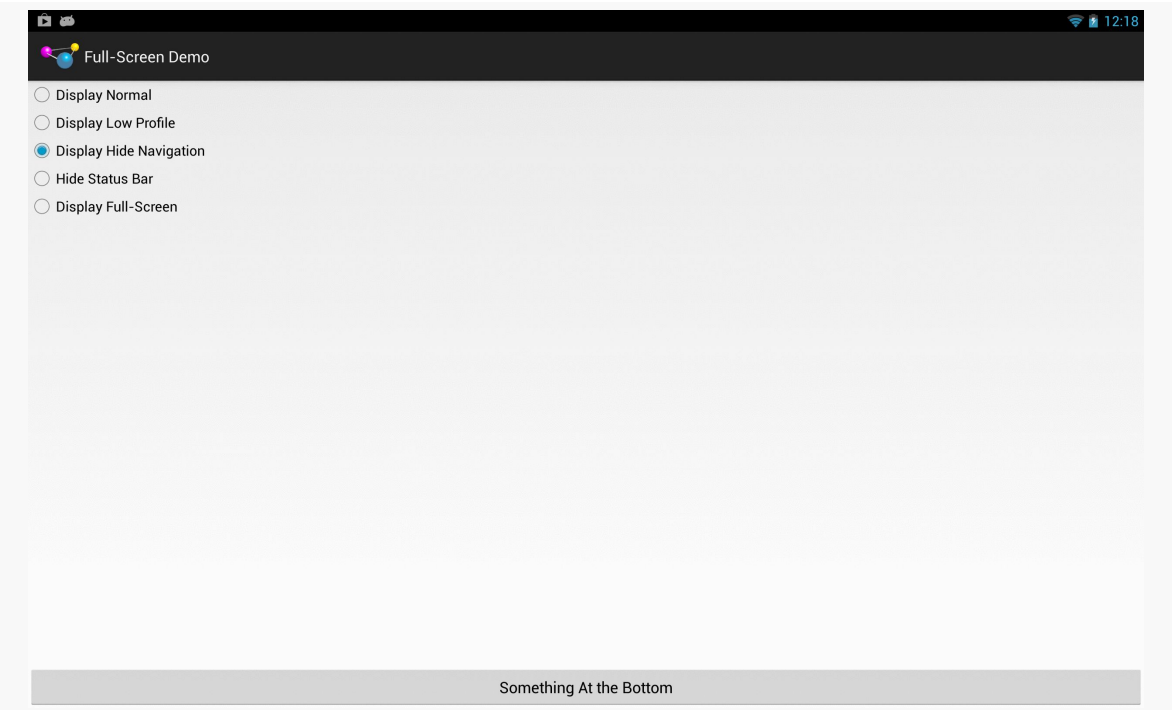


Figure 564: Sample UI, Hidden-Navigation Mode, Nexus 10/Android 4.2

Similarly, you can hide the status bar by passing `View.SYSTEM_UI_FLAG_FULLSCREEN` to `setSystemUiVisibility()`. However, despite this flag's name, it does not affect the navigation or system bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

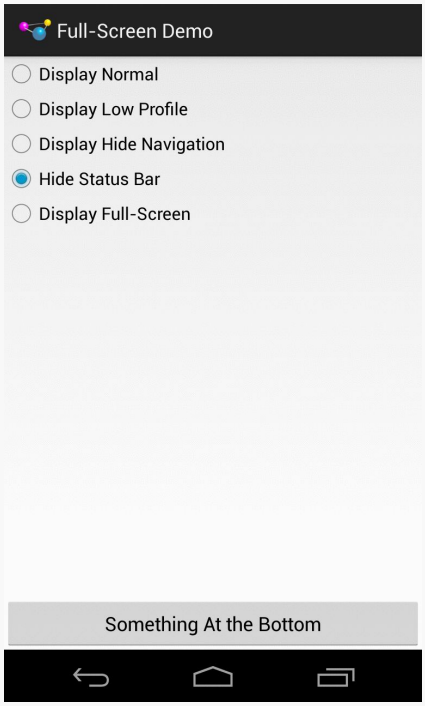


Figure 565: Sample UI, "Full-Screen" Mode, Nexus 4/Android 4.2

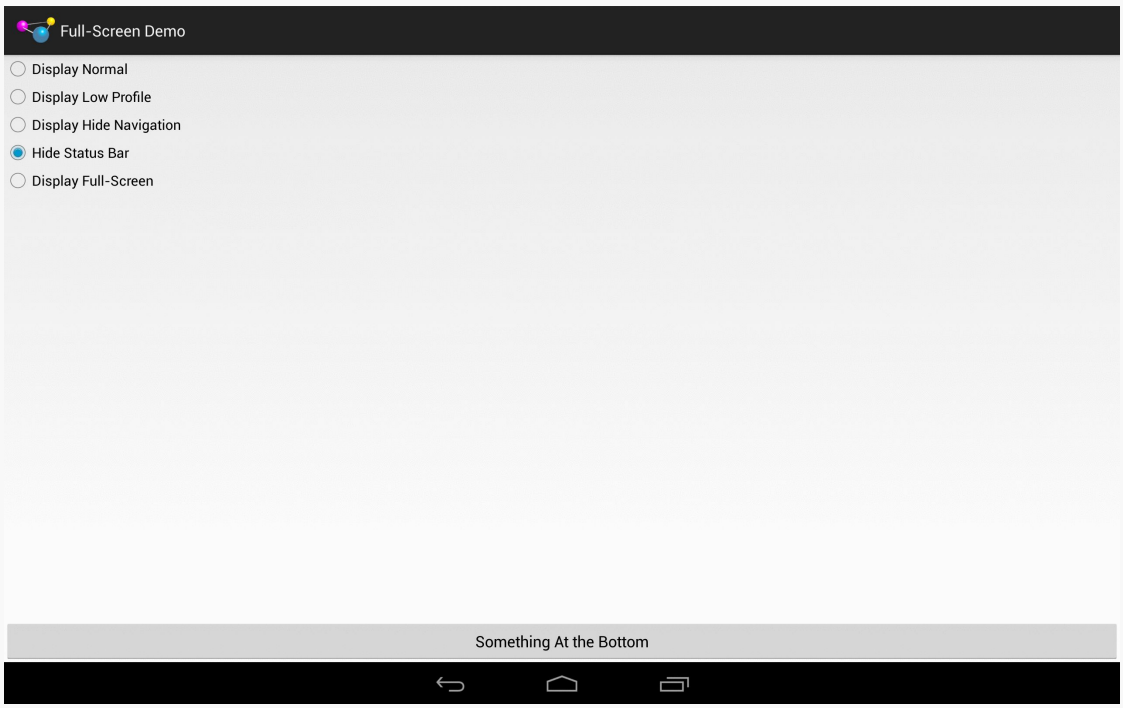


Figure 566: Sample UI, "Full-Screen" Mode, Nexus 10/Android 4.2

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Hence, to hide both the status bar and the navigation or system bar, you need to pass both flags (`View.SYSTEM_UI_FLAG_FULLSCREEN | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION`):

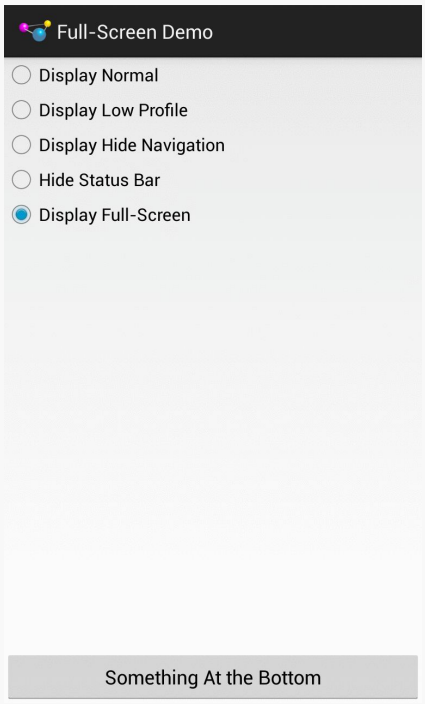


Figure 567: Sample UI, True Full-Screen Mode, Nexus 4/Android 4.2

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 568: Sample UI, True Full-Screen Mode, Nexus 10/Android 4.2

Note that showing and hiding the `ActionBar` is also possible, via calls to `show()` and `hide()`, respectively.

Offering a Delayed Timeout

Android makes it easy for activities to keep the screen on while the activity is in the foreground, by means of `android:keepScreenOn` and `setKeepScreenOn()`.

However, these are very blunt instruments, and too many developers simply ask to keep the screen on constantly, even when that is not needed and can cause excessive battery drain. That is because it is very easy to *always* keeps the screen on.

Say, for example, you are playing a game. Keeping the screen on while the game is being played is probably a good thing, particularly if the game does not require constant interaction with the screen. However, if you press the in-game pause button, the game might keep the screen on while the game is paused. This might lead you to press pause, put down your tablet (expecting it to fall asleep in a normal period of time), and then have the tablet keep going and going and going... until the battery runs dead.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Whether you use `setKeepScreenOn()` or directly use a `WakeLock`, it is useful to think of three tiers of user interaction.

The first tier is when your app is doing its "one big thing": playing the game, playing the video, displaying the digital book, etc. If you expect that there will be periods of time when the user is actively engaged with your app, but is not interacting with the screen, keep the screen on.

The second tier is when your app is delivering something to the user that *probably* would get used without interaction in the short term, but not indefinitely. For example, a game might reasonably expect that 15 seconds could be too short to have the screen time out, but if the user has not done anything in 5–10 minutes, most likely they are not in front of the game. Similarly, a digital book reader should not try to keep the screen on for an hour without user interaction.

The third tier is when your app is doing anything other than the main content, where normal device behavior should resume. A video player might keep the screen on while the video is playing, but if the video ends, normal behavior should resume. After all, if the person who had been watching the video fell asleep, they will not be in position to press a power button.

The first and third tiers are fairly easy from a programming standpoint. Just `acquire()` and `release()` the `WakeLock`, or toggle `setKeepScreenOn()` between `true` and `false`.

The second tier — where you are willing to have a screen timeout, just not too quickly — requires you to add a bit more smarts to your app. A simple, low-overhead way of addressing this is to have a `postDelayed()` loop, to get a `Runnable` control every 5–10 seconds. Each time the user interacts with your app, update a `lastInteraction` timestamp. The `Runnable` compares `lastInteraction` with the current time, and if it exceeds some threshold, release the `WakeLock` or call `setKeepScreenOn(false)`. When the user interacts again, though, you will need to re-acquire the `WakeLock` or call `setKeepScreenOn(true)`. Basically, you have your *own* inactivity timing mechanism to control when you are inhibiting normal inactivity behavior or not.

To see the second tier in action, take a look at the `MiscUI/DelayedTimeout` sample project.

The UI is a simple button. We want to keep the screen awake while the user is using the button, but let it fall asleep after a period of inactivity that *we* control. To

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

accomplish this, we will use a `postDelayed()` loop, to get control every 15 seconds to see if there has been user activity:

```
package com.commonsware.android.timeout;

import android.app.Activity;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;

public class MainActivity extends Activity implements Runnable {
  private static int TIMEOUT_POLL_PERIOD=15000; // 15 seconds
  private static int TIMEOUT_PERIOD=300000; // 5 minutes
  private View content=null;
  private long lastActivity=SystemClock.uptimeMillis();

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    content=findViewById(android.R.id.content);
    content.setKeepScreenOn(true);
    run();
  }

  @Override
  public void onDestroy() {
    content.removeCallbacks(this);

    super.onDestroy();
  }

  @Override
  public void run() {
    if ((SystemClock.uptimeMillis() - lastActivity) > TIMEOUT_PERIOD) {
      content.setKeepScreenOn(false);
    }

    content.postDelayed(this, TIMEOUT_POLL_PERIOD);
  }

  public void onClick(View v) {
    lastActivity=SystemClock.uptimeMillis();
  }
}
```

In `onCreate()`, we call `setKeepScreenOn(true)` to keep the screen on, regardless of what the user's default timeout is. Then, we call the `run()` method from our `Runnable` interface (implemented on the activity itself). `run()` sees if 5 minutes has elapsed since the last bit of user activity (initially set to be the time the activity

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

launches). If 5 minutes has elapsed, we revert to normal screen-timeout behavior with `setKeepScreenOn(false)`. We also schedule ourselves, as a `Runnable`, to get control again in 15 seconds, to see if 5 minutes has elapsed since the last-seen activity. Our button's `onClick()` method simply updates the last-seen timestamp, and `onDestroy()` cleans up our `postDelayed()` loop by calling `removeCallbacks()` to stop invoking our `Runnable`.

The net is that the device's screen will remain on for 5 minutes since the last time the user taps the button, even if the user's default screen timeout is set to shorter than 5 minutes. Yet, at the same time, we do not keep the screen on *forever*, causing unnecessary battery drain.

Note that to test this, you will probably need to unplug your USB cable after installing the app on the device (since many developers have it set up to keep the screen on while plugged in). Also, you will need to set your device's screen timeout to be under 5 minutes, if it is not set that way already.

This is a primitive implementation, missing lots of stuff that you would want in production code (e.g., it never calls `setKeepScreenOn(true)` if we flipped it to `false` but *then* tap the button). And the complexity of determining if the user interacted with the screen will be tied to the complexity of your UI.

That being said, by having a more intelligent use of `WakeLock` and `setKeepScreenOn()`, you can deliver value to the user while not accidentally causing excessive battery drain. Users do not always remember to press the power button, so you need to make sure that just because the user made a mistake, that you do not make it worse.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Event Buses Alternatives

Earlier in the book, we covered the concept of an event bus as a way of communicating between portions of our app, focusing on one event bus implementation: greenrobot's EventBus. Later, in the chapter on broadcast `Intents`, we briefly covered `LocalBroadcastManager`.

However, those are not the only event buses available for Android, and others may fit your needs better. In this chapter, we will explore these and other event bus implementations, to compare and contrast.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the chapters on basic event bus usage, broadcast `Intents`, `AlarmManager` and the scheduled service pattern, and `Notifications`.

A Brief Note About the Sample Apps

The sample apps in this chapter are generally designed to run forever.

It is unlikely that you really want them to run forever, though. Hence, please uninstall each sample after experimenting with it, particularly if you are testing on hardware, such as your personal phone. Your battery will appreciate it.

Standard Intents as Event Bus

You can think of the standard `Intent` and `<intent-filter>` system as a three-channel event bus:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- One channel is used for starting activities
- One channel is used for starting or binding to services
- One channel is used for more ad-hoc "broadcast" events

The component starting an activity does not need to communicate directly with code for that activity — in fact, often times this is impossible, as they are separate apps running in separate processes. Instead, the component starting an activity sends an event indicating the particular operation to be performed (e.g., view this URL), and Android and the user determine which of candidate consumers is the one to process that event.

However, broadcast `Intents` are a closer analogue to a real "event bus", in that an event produced by somebody can be consumed by zero, one, or several subscribed consumers, based upon the filtering provided by `<intent-filter>` elements in the manifest or `IntentFilter` objects for use with `registerReceiver()`.

In theory, you could use broadcast `Intents` as the backbone for a fairly flexible event bus within your app. In practice, this is not usually a good idea:

- Each broadcast involves inter-process communication (IPC), even if the event producer and consumer(s) are in the same process. This adds overhead.
- Because broadcasts are intrinsically IPC, you have to take security into account, to ensure only authorized producers can publish events that the consumers pick up.

LocalBroadcastManager as Event Bus

As was briefly noted earlier in the book, the Android Support package offers a `LocalBroadcastManager`. This is designed to offer an event bus with a feel very similar to classic broadcast `Intents`, but local to your process. Not only does this avoid IPC overhead, but it improves security, as other apps have no means of spying on your internal communications.

`LocalBroadcastManager` is supplied by both the `android-support-v4.jar` and `android-support-v13.jar` libraries. Generally speaking, if your `android:minSdkVersion` is less than 13, you probably should choose `android-support-v4.jar`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A Simple LocalBroadcastManager Sample

Let's see `LocalBroadcastManager` in action via the Intents/Local sample project.

Here, our `LocalActivity` sends a command to a `NoticeService` from `onCreate()`:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    notice=(TextView)findViewById(R.id.notice);
    startService(new Intent(this, NoticeService.class));
  }
```

The `NoticeService` simply delays five seconds, then sends a local broadcast using `LocalBroadcastManager`:

```
package com.commonsware.android.localcast;

import android.app.IntentService;
import android.content.Intent;
import android.os.SystemClock;
import android.support.v4.content.LocalBroadcastManager;

public class NoticeService extends IntentService {
  public static final String BROADCAST=
      "com.commonsware.android.localcast.NoticeService.BROADCAST";
  private static Intent broadcast=new Intent(BROADCAST);

  public NoticeService() {
    super("NoticeService");
  }

  @Override
  protected void onHandleIntent(Intent intent) {
    SystemClock.sleep(5000);
    LocalBroadcastManager.getInstance(this).sendBroadcast(broadcast);
  }
}
```

Specifically, you get at your process' singleton instance of `LocalBroadcastManager` by calling `getInstance()` on the `LocalBroadcastManager` class.

Our `LocalActivity` registers for this local broadcast in `onResume()`, once again using `getInstance()` on `LocalBroadcastManager`:

```
  @Override
  public void onResume() {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onResume();

    IntentFilter filter=new IntentFilter(NoticeService.BROADCAST);

    LocalBroadcastManager.getInstance(this).registerReceiver(onNotice,
                                                             filter);
  }
```

`LocalActivity` unregisters for this broadcast in `onPause()`:

```
  @Override
  public void onPause() {
    super.onPause();

    LocalBroadcastManager.getInstance(this).unregisterReceiver(onNotice);
  }
```

The `BroadcastReceiver` simply updates a `TextView` with the current date and time:

```
  private BroadcastReceiver onNotice=new BroadcastReceiver() {
    public void onReceive(Context ctxt, Intent i) {
      notice.setText(new Date().toString());
    }
  };
```

If you start up this activity, you will see a "`(waiting...)`" bit of placeholder text for about five seconds, before having that be replaced by the current date and time.

The `BroadcastReceiver`, the `IntentFilter`, and the `Intent` being broadcast are the same as we would use with full broadcasts. It is merely how we are using them — via `LocalBroadcastManager` – that dictates they are local to our process versus the standard device-wide broadcasts.

A More Elaborate Sample

That sample is not terribly realistic, but it is simple.

A somewhat more realistic sample is the one using `WakefulIntentService` from earlier in the book. However, that app was also fairly unrealistic, at least in terms of its output, as LogCat is not very useful to users. A more typical approach for a background service like this is to notify a foreground `Activity`, if there is one, about work that was accomplished, and otherwise display a `Notification`. We described that pattern in the chapter on `Notifications`.

In the `EventBus/LocalBroadcastManager` sample project, we blend:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Having a service wake up every so often to do some work
- Arranging to let the user know of background accomplishments via an `Activity` or a `Notification`
- Using `LocalBroadcastManager` to keep the communications in-process

The Activity

The `EventDemoActivity` that is our app's entry point is a bit similar to the one used in the `WakefulIntentService` demo, in that it calls `scheduleAlarms()` on `PollReceiver` to set up the `AlarmManager` schedule:

```
package com.commonsware.android.eventbus.lbm;

import android.app.Activity;
import android.os.Bundle;

public class EventDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new EventLogFragment()).commit();

      PollReceiver.scheduleAlarms(this);
    }
  }
}
```

However, we also put an `EventLogFragment` on the screen, if it is not already there, via a `FragmentTransaction`. This is where we will display events coming from the service, while our activity is in the foreground. We will examine `EventLogFragment` and how it participates in the event bus shortly.

The PollReceiver

`PollReceiver` is unchanged from its `WakefulIntentService` demo original edition. This `BroadcastReceiver` will be used both for getting control at boot time (to reschedule the alarms, wiped on the reboot) and for sending the work to the `ScheduledService` for processing:

```
package com.commonsware.android.eventbus.lbm;

import android.app.AlarmManager;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class PollReceiver extends BroadcastReceiver {
  private static final int PERIOD=15000; // 15 seconds
  private static final int INITIAL_DELAY=1000; // 1 second

  @Override
  public void onReceive(Context ctxt, Intent i) {
    if (i.getAction() == null) {
      WakefulIntentService.sendWakefulWork(ctxt, ScheduledService.class);
    }
    else {
      scheduleAlarms(ctxt);
    }
  }

  static void scheduleAlarms(Context ctxt) {
    AlarmManager mgr=
        (AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE);
    Intent i=new Intent(ctxt, PollReceiver.class);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);

    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                     SystemClock.elapsedRealtime() + INITIAL_DELAY,
                     PERIOD, pi);

  }
}
```

ScheduledService and Sending Events

Before, our `ScheduledService` just dumped a message to LogCat. This was crude but effective for what that demo required. Now, we want our service to let the UI layer know about some work that was accomplished, or to raise a `Notification`.

In this case, the "work" is generating a random number.

```
package com.commonsware.android.eventbus.lbm;

import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.support.v4.app.NotificationCompat;
import android.support.v4.content.LocalBroadcastManager;
import java.util.Calendar;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import java.util.Random;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class ScheduledService extends WakefulIntentService {
  private static int NOTIFY_ID=1337;
  private Random rng=new Random();

  public ScheduledService() {
    super("ScheduledService");
  }

  @Override
  protected void doWakefulWork(Intent intent) {
    Intent event=new Intent(EventLogFragment.ACTION_EVENT);
    long now=Calendar.getInstance().getTimeInMillis();
    int random=rng.nextInt();

    event.putExtra(EventLogFragment.EXTRA_RANDOM, random);
    event.putExtra(EventLogFragment.EXTRA_TIME, now);

    if (!LocalBroadcastManager.getInstance(this).sendBroadcast(event)) {
      NotificationCompat.Builder b=new NotificationCompat.Builder(this);
      Intent ui=new Intent(this, EventDemoActivity.class);

      b.setAutoCancel(true).setDefaults(Notification.DEFAULT_SOUND)
       .setContentTitle(getString(R.string.notif_title))
       .setContentText(Integer.toHexString(random))
       .setSmallIcon(android.R.drawable.stat_notify_more)
       .setTicker(getString(R.string.notif_title))
       .setContentIntent(PendingIntent.getActivity(this, 0, ui, 0));

      NotificationManager mgr=
          (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

      mgr.notify(NOTIFY_ID, b.build());
    }
  }
}
```

`LocalBroadcastManager`, as we have seen, uses the same `Intent` and `IntentFilter` and `BroadcastReceiver` structures as are used with regular broadcasts, just via a singleton message bus (`LocalBroadcastManager.getInstance()`) instead of the framework's IPC engine. Hence, we need an `Intent` that represents the message, so we create one, using an action string published by the `EventLogFragment`. We also attach two extras to this `Intent`, using keys published by `EventLogFragment`: the random number, plus the time of this event.

We then call `sendBroadcast()` on the singleton `LocalBroadcastManager`. This returns a `boolean` value, `true` indicating that one or more locally-registered receivers were delivered the `Intent`, `false` otherwise. Hence, if `sendBroadcast()` returns

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`true`, we can assume that somebody in the UI layer picked up our message and is now responsible for displaying these results to the user.

Conversely, if `sendBroadcast()` returns `false`, we must assume that the UI layer did not receive the message, and so the service should inform the user directly, in this case via a `Notification`, showing the random number as the text in the notification drawer.

EventLogFragment and Receiving Events

`EventLogFragment`, therefore, is responsible for:

- Registering (and unregistering) to receive the broadcasts to be sent locally by the service
- Doing something with those events to inform the user about the all-important random numbers

In this case, we use a retained `ListFragment` with a `ListView` set into transcript mode, meaning that entries are added at the bottom, and older entries scroll off the top, like a chat transcript:

```
package com.commonsware.android.eventbus.lbm;

import android.annotation.SuppressLint;
import android.app.ListFragment;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.support.v4.content.LocalBroadcastManager;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.Locale;

public class EventLogFragment extends ListFragment {
  static final String EXTRA_RANDOM="r";
  static final String EXTRA_TIME="t";
  static final String ACTION_EVENT="e";
  private EventLogAdapter adapter=null;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@Override
public void onActivityCreated(Bundle state) {
  super.onActivityCreated(state);

  setRetainInstance(true);
  getListView().setTranscriptMode(ListView.TRANSCRIPT_MODE_NORMAL);

  if (adapter == null) {
    adapter=new EventLogAdapter();
  }

  setListAdapter(adapter);
}

@Override
public void onResume() {
  super.onResume();

  IntentFilter filter=new IntentFilter(ACTION_EVENT);

  LocalBroadcastManager.getInstance(getActivity())
                       .registerReceiver(onEvent, filter);
}

@Override
public void onPause() {
  LocalBroadcastManager.getInstance(getActivity())
                       .unregisterReceiver(onEvent);

  super.onPause();
}

class EventLogAdapter extends ArrayAdapter<Intent> {
  DateFormat fmt=new SimpleDateFormat("HH:mm:ss", Locale.US);

  public EventLogAdapter() {
    super(getActivity(), android.R.layout.simple_list_item_1,
          new ArrayList<Intent>());
  }

  @SuppressLint("DefaultLocale")
  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    TextView row=
        (TextView)super.getView(position, convertView, parent);
    Intent event=getItem(position);
    Date date=new Date(event.getLongExtra(EXTRA_TIME, 0));

    row.setText(String.format("%s = %x", fmt.format(date),
                              event.getIntExtra(EXTRA_RANDOM, -1)));

    return(row);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }

  private BroadcastReceiver onEvent=new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
      adapter.add(intent);
    }
  };
}
```

The `ListAdapter` for the `ListView` is an `EventLogAdapter`, an `ArrayAdapter` for `Intent` objects, where in `getView()` we populate the list rows with the time and random value.

In `onResume()` and `onPause()`, we register for (and unregister from) the desired broadcast, pointing to an `onEvent` `BroadcastReceiver` that adds the incoming `Intent` to the `EventLogAdapter`. That, in turn, updates the `ListView`.

The result is that while the activity is in the foreground, the events will be displayed to the user directly:

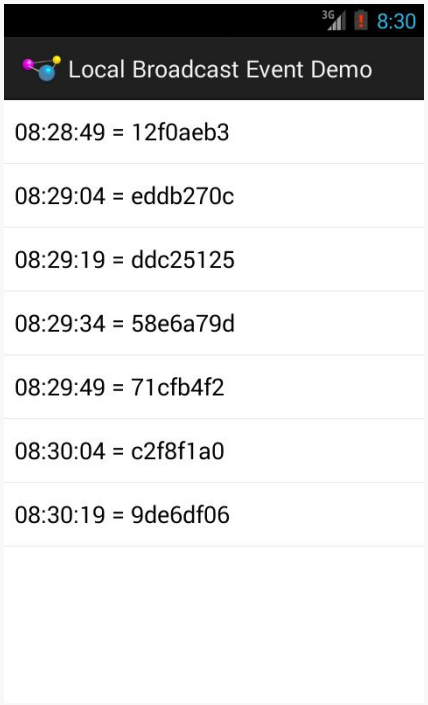


Figure 569: LocalBroadcastManager as Event Bus, Demo Activity

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Whereas if events are processed while the activity is not in the foreground, a `Notification` will be shown with the last results:

Figure 570: LocalBroadcastManager as Event Bus, Demo Notification

Reference, Not Value

When you send a "real" broadcast `Intent`, your `Intent` is converted into a byte array (courtesy of the `Parcelable` interface) and transmitted to other processes. This occurs even if the recipient of the `Intent` is within your own process — that is what makes `LocalBroadcastManager` faster, as it avoids the inter-process communication.

However, since `LocalBroadcastManager` does not need to send your Intent between processes, that means it does not turn your `Intent` into a byte array. Instead, it just passes the `Intent` along to any registered `BroadcastReceiver` with a matching `IntentFilter`. In effect, while "real" broadcasts are pass-by-value, local broadcasts are pass-by-reference.

This can have subtle side effects.

For example, there are a few ways that you can put a collection into an `Intent` extra, such as `putStringArrayListExtra()`. This takes an `ArrayList` as a parameter. With

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

a real broadcast, once you send the broadcast, it does not matter what happens to the original `ArrayList` — the rest of the system is working off of a copy. With a local broadcast, though, the `Intent` holds onto the `ArrayList` you supplied via the setter. If you change that `ArrayList` elsewhere (e.g., clear it for reuse), the recipient of the Intent will see those changes.

Similarly, if you put a `Parcelable` object in an extra, the `Intent` holds onto the actual object while it is being broadcast locally, whereas a real broadcast would have resulted in a copy. If you change the object while the broadcast is in progress, the recipient of the broadcast will see those changes.

This can be a feature, not a bug, when used properly. But, regardless, it is a non-trivial difference, one that you will need to keep in mind.

Limitations of Local

While `LocalBroadcastManager` is certainly useful, it has some serious limitations.

The biggest is that it is purely local. While traditional broadcasts can either be internal (via `setPackage()`) or device-wide, `LocalBroadcastManager` only handles the local case. Hence, anything that might involve other processes, such as a `PendingIntent`, will not use `LocalBroadcastManager`. For example, you cannot register a receiver through `LocalBroadcastManager`, then use a `getBroadcast()` `PendingIntent` to try to reach that `BroadcastReceiver`. The `PendingIntent` will use the regular broadcast `Intent` mechanism, which the local-only receiver will not respond to.

Similarly, since a manifest-registered `BroadcastReceiver` is spawned via the operating system upon receipt of a matching true broadcast, you cannot use such receivers with `LocalBroadcastManager`. Only a `BroadcastReceiver` registered via `registerReceiver()` on the `LocalBroadcastManager` will use the `LocalBroadcastManager`. For example, you cannot implement the `Activity`-or-`Notification` pattern that we will see later in this book via `LocalBroadcastManager`.

Also, `LocalBroadcastManager` does not offer ordered or sticky broadcasts.

Square's Otto

`LocalBroadcastManager` has two major advantages:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. It is part of the Android Support package, and therefore it is part of the officially-supported corner of the Android ecosystem
2. It works like traditional broadcasts, which will make it easier for some developers to "wrap their heads around" it

However, that same dependency on the `Intent` and `IntentFilter` structure adds bulk and limits flexibility. Hence, it is not surprising that there are alternative event buses to `LocalBroadcastManager`.

Java, outside of Android, has had a few event bus implementations. One of the more popular ones in recent years has been the event bus that is part of Google's Guava family of libraries. However, while a Java event bus perhaps can be used on Android, it may not be *optimal* for Android. Hence, a few projects have started with Guava's event bus implementation and have extended it to be a bit more Android-aware, or perhaps even Android-centric.

Square's Otto is one such event bus.

Basic Usage and Sample App

With `LocalBroadcastManager`, you work with a singleton instance, calling methods like `registerReceiver()` and `sendBroadcast()` upon it to subscribe to and raise events, respectively.

With Otto, you work with a singleton instance of a `Bus`, calling methods like `register()` and `post()` upon it to subscribe to and raise events, respectively.

Hence, at the core, Otto behaves much like `LocalBroadcastManager`. What differs is in the nature of the events and the subscribers.

With `LocalBroadcastManager`, events are `Intents`. With Otto, an event can be whatever data type you like. Hence, you can create your own `...Event` classes, holding whatever bits of data, in whatever data types suit you — you are not restricted to things that can go in an `Intent` extra. However, as has been noted on occasion, "with great power comes great responsibility", and so you will need to ensure that you use this carefully and do not wind up creating some sort of memory leak as a result. For example, do not pass something from an `Activity` to a `Service` via a custom event, where the `Service` will hold onto that information for a long time, if that "something" holds a reference back to the `Activity`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

With LocalBroadcastManager, subscribers are BroadcastReceivers, who use an IntentFilter to identify which events they are interested in. With Otto, subscribers are any class you want. A special @Subscribe annotation is used to both indicate what sorts of events the subscriber is interested in (based on the parameter to the annotated method) and what method should be invoked when a matching event is raised (the annotated method itself). Hence, not only do you use custom event classes to allow you to carry along custom data, but you use them as a filtering mechanism, much like you would use custom action strings with LocalBroadcastManager.

To see how this works, take a look at the EventBus/Otto sample project, which is a clone of the EventBus/LocalBroadcastManager demo, but one where we substitute in Otto as a replacement for LocalBroadcastManager. Our activity and PollReceiver are unchanged: they did not directly interact with LocalBroadcastManager and do not need to interact with Otto. The changes are isolated in our ScheduledService and EventLogFragment.

ScheduledService and Sending Events

First, we need a singleton Otto Bus instance, that serves the same basic role as does the singleton LocalBroadcastManager retrieved by getInstance(). However, Otto does not offer a similar getInstance() method. It is up to you to create and manage your own singleton. This gives you greater flexibility, such as having multiple independent buses for disparate event channels. However, it does mean that you need to put a Bus somewhere.

Fortunately, Bus does not need a Context, and so we can initialize a singleton as a static data member somewhere. Here, the "somewhere" is on ScheduledService:

```
  static final Bus bus=new Bus(ThreadEnforcer.ANY);
```

The parameter to the Bus constructor is the threading rule to be enforced on this bus. If you attempt to use the Bus on a thread that is disallowed by the supplied ThreadEnforcer, you will get an IllegalStateException at runtime. ThreadEnforcer.MAIN ensures that you only use the Bus on Android's main application thread. ThreadEnforcer.ANY allows the use of the Bus on any thread... though then you will need to do your own work to route control back to the main application thread, if needed.

Now, when it comes time for us to send a message, we can call post() on the Bus, supplying whatever sort of event object that we want:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  protected void doWakefulWork(Intent intent) {
    bus.post(new RandomEvent(rng.nextInt()));
  }
```

Here, we are posting an instance of a `RandomEvent`:

```
package com.commonsware.android.eventbus.otto;

import java.util.Calendar;
import java.util.Date;

public class RandomEvent {
  Date when=Calendar.getInstance().getTime();
  int value;

  RandomEvent(int value) {
    this.value=value;
  }
}
```

EventLogFragment and Receiving Events

Over in our `EventLogFragment`, rather than register and unregister a `BroadcastReceiver` in `onResume()` and `onPause()`, we register and unregister the fragment itself with the singleton `Bus`:

```
  @Override
  public void onResume() {
    super.onResume();

    ScheduledService.bus.register(this);
  }

  @Override
  public void onPause() {
    ScheduledService.bus.unregister(this);

    super.onPause();
  }
```

Now, we can use the `@Subscribe` annotation to arrange to receive any event we want that is delivered via this `Bus`, based on event class. Since we want to receive `RandomEvent` messages, we merely need to have a `public void` method, taking a `RandomEvent` parameter, marked with the `@Subscribe` annotation, such as `onRandomEvent()`:

```
  @Subscribe
  public void onRandomEvent(final RandomEvent event) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (getActivity() != null) {
      getActivity().runOnUiThread(new Runnable() {
        @Override
        public void run() {
          adapter.add(event);
        }
      });
    }
  }
```

Note that the method name can be anything we want, as it is the annotation, not the method name, that identifies this as being an event handling method.

In this method, we can do what we need to with our `RandomEvent`. In our case, `EventLogAdapter` has been modified to be an `ArrayAdapter` of `RandomEvent`, as opposed to being an `ArrayAdapter` of `Intent` as in the earlier sample. What we want to do is append the new `RandomEvent` to the end of the adapter.

However, while `LocalBroadcastManager` will only deliver events on the main application thread, Otto delivers events *on whatever thread they were sent upon*. In this case, we know that this will be a background thread, the one used by the `IntentService`. We cannot safely modify the `EventLogAdapter` on a background thread, as that will update the UI. So, we need to call `add()` on the adapter on the main application thread. Here, we use `runOnUiThread()` to pass a `Runnable` to the main application thread containing our `add()` call. However, it is possible that we do not have an activity right this moment, such as due to a configuration change. This demo simply drops those events; a production-grade app might wish to queue those within the fragment and apply them in some later lifecycle event (e.g., `onAttach()`, `onActivityCreated()`).

Handling the "Nobody's Home" Scenario

What is missing, though, is the logic we used in `LocalBroadcastManager` to determine if somebody received our message, where we raised a `Notification` if that is not the case.

The solution for this with Otto is to have `ScheduledService` listen for `DeadEvent` events. A `DeadEvent` is delivered on the bus when an attempt to deliver some other event failed with no subscribers. The `DeadEvent` has an `event` field that contains the original event that failed to be delivered. If we can get the `DeadEvent`, we know the `RandomEvent` was not handled at the UI layer, and we can raise the `Notification`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To do this, not only do we need to register `EventLogFragment` with the `Bus`, but we also need to register `ScheduledService` itself, so it can listen for a `DeadEvent`:

```
@Override
public void onCreate() {
  super.onCreate();

  bus.register(this);
}
```

```
@Override
public void onDestroy() {
  bus.unregister(this);

  super.onDestroy();
}
```

Then, our `Notification` logic can be moved into some method that has the `@Subscribe` annotation and a `DeadEvent` parameter:

```
@Subscribe
public void onDeadEvent(DeadEvent braiiiiiiinz) {
  RandomEvent original=(RandomEvent)braiiiiiiinz.event;
  NotificationCompat.Builder b=new NotificationCompat.Builder(this);
  Intent ui=new Intent(this, EventDemoActivity.class);

  b.setAutoCancel(true).setDefaults(Notification.DEFAULT_SOUND)
   .setContentTitle(getString(R.string.notif_title))
   .setContentText(Integer.toHexString(original.value))
   .setSmallIcon(android.R.drawable.stat_notify_more)
   .setTicker(getString(R.string.notif_title))
   .setContentIntent(PendingIntent.getActivity(this, 0, ui, 0));

  NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

  mgr.notify(NOTIFY_ID, b.build());
}
```

Event Producers

Standard broadcasts in Android can be broadcast in a "sticky" fashion. When an app registers for a sticky broadcast, not only does the app get any future matching broadcasts, but immediately it gets the last-broadcast `Intent` that matches.

`LocalBroadcastManager` does not offer a similar capability, and neither does Otto in the direct sense. However, Otto does have the concept of event producers, which can play a similar role.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You can annotate a method with `@Produce`, to indicate that it is an event producer. The method should take no parameters, but its return type should be one of your event classes (e.g., `RandomEvent`).

When an app calls `register()` on a `Bus`, Otto finds all of the `@Subscribe`-annotated methods and keeps track of them, for dispatching future events. However, in addition, Otto also checks to see if there is a matching event producer for the event type requested by the `@Subscribe` method. If there is a matching producer, the producer method is called, and that event is passed to the subscriber immediately, as part of the `register()` processing.

So, if we had a `@Produce` method in the sample app that returned a `RandomEvent`, the `onRandomEvent()` method would have been invoked immediately with the result of calling that `@Produce` method, in addition to being called with any future events raised by the app.

This is useful for cases where there may be interruptions in event processing. For example, in a configuration change, your activity and fragments are destroyed and recreated by default. But if you are posting events on a background thread, as we did in the sample, those events could occur while a configuration change is in process, and there may not be an available subscriber for the event right at that moment. The `@Produce` pattern would allow you to cache that result and give it to the new activity or fragment.

Revisiting greenrobot's EventBus

Of the three major in-process event bus implementations, greenrobot's EventBus is slightly less popular than `LocalBroadcastManager` and Otto. `LocalBroadcastManager` is part of the Android SDK (albeit in the Android Support package), and so it will gain popularity from that status alone. Square has a vast range of libraries and has a following simply from all the work they have done, plus the work of individual Square engineers (e.g., Jake Wharton, author of ActionBarSherlock, ViewPagerIndicator, and many others).

This is not to say that greenrobot's EventBus is weaker than the alternatives. In fact, it may be the most powerful of the three. Like Otto, greenrobot's EventBus has Guava's EventBus in its heritage. Unlike Otto, greenrobot eschewed annotations, opting instead for a convention-based system utilizing method name patterns, to avoid the overhead of runtime annotations on Android prior to 4.0.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The EventBus/GreenRobot sample project is a clone of the EventBus/Otto project, replacing Otto with greenrobot's EventBus, that we will examine in this section.

Basic Usage and Sample App

As both Otto and greenrobot's EventBus have Guava's EventBus as antecedents, the flow of using both is similar, just with differing details.

ScheduledService

ScheduledService has a very similar feel to its Otto equivalent:

```
package com.commonsware.android.eventbus.greenrobot;

import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.support.v4.app.NotificationCompat;
import java.util.Random;
import com.commonsware.cwac.wakeful.WakefulIntentService;
import de.greenrobot.event.EventBus;
import de.greenrobot.event.NoSubscriberEvent;

public class ScheduledService extends WakefulIntentService {
  private static int NOTIFY_ID=1337;
  private Random rng=new Random();

  public ScheduledService() {
    super("ScheduledService");
  }

  @Override
  public void onCreate() {
    super.onCreate();

    EventBus.getDefault().register(this);
  }

  @Override
  protected void doWakefulWork(Intent intent) {
    EventBus.getDefault().post(new RandomEvent(rng.nextInt()));
  }

  @Override
  public void onDestroy() {
    EventBus.getDefault().unregister(this);

    super.onDestroy();
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  public void onEvent(NoSubscriberEvent event) {
    RandomEvent randomEvent=(RandomEvent)event.originalEvent;
    NotificationCompat.Builder b=new NotificationCompat.Builder(this);
    Intent ui=new Intent(this, EventDemoActivity.class);

    b.setAutoCancel(true).setDefaults(Notification.DEFAULT_SOUND)
     .setContentTitle(getString(R.string.notif_title))
     .setContentText(Integer.toHexString(randomEvent.value))
     .setSmallIcon(android.R.drawable.stat_notify_more)
     .setTicker(getString(R.string.notif_title))
     .setContentIntent(PendingIntent.getActivity(this, 0, ui, 0));

    NotificationManager mgr=
        (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

    mgr.notify(NOTIFY_ID, b.build());
  }
}
```

There are only two significant differences.

First, we use `EventBus.getDefault()` to get the stock `EventBus` singleton, rather than create our own instance as we did with Otto. We *could* create our own instance, though, as opposed to `LocalBroadcastManager`, which seems to prefer its own singleton.

Second, instead of having an `onDeadEvent()` `@Subscribe` method, we register our own interest in EventBus events, by calling `register()` on the `EventBus` instance in `onCreate()`.

The service itself then has an `onEvent()` method for our `NoSubscriberEvent`, where it raises the `Notification`.

When using greenrobot's EventBus, rather than using annotations to denote event-handling methods, we use a naming scheme: `onEvent()` (or variations on that theme, as will be seen shortly). Since Java allows method overloading with different parameter lists, this works just fine — we can have N event-handling methods in a class, with N different event types as the parameter. The greenrobot code simply finds all of the event-handling methods by name, rather than by annotation. This does limit the flexibility in choosing method names, though.

The `NoSubscriberEvent` fills the same basic role as the `DeadEvent` in Otto – it is raised when somebody else raises an event that nobody subscribed to. In our case, as with Otto's `DeadEvent`, we are using `NoSubscriberEvent` to determine when the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`RandomEvent` is not picked up by the `EventLogFragment`, so we can raise the `Notification`.

EventLogFragment

The modified `EventLogFragment` also uses `EventBus.getDefault()` to register and unregister from the default bus instance. It too has a renamed event-handling method, now called `onEventMainThread()`:

```
public void onEventMainThread(final RandomEvent event) {
  adapter.add(event);
}
```

The default behavior of threading with greenrobot's EventBus is to deliver the event on the same thread that posted the event, much like Otto. The `onEvent()` method used in `ScheduledService`, for example, accepts the event on the posting thread. However, there are other variations of the `onEvent()` method name that you can use, to signify that you want to have the event delivered on some other thread. `onEventMainThread()` indicates that we want to have the event delivered to the method on the main application thread, which is why the `EventLogFragment` can safely update the UI.

Hence, while the fragment is in the foreground, it will handle the `RandomEvent`; otherwise, the service will.

Other Notable Capabilities

In addition to the threading features, greenrobot's EventBus has a few other noteworthy bells and whistles:

- `register()` and `unregister()` optionally take an event class (e.g., `RandomEvent.class`) as a second parameter, allowing you to register and unregister from specific events, in addition to the default behavior of registering for all events for which you have handlers.
- `postSticky()` and `registerSticky()` allow you to have sticky events, much like sticky broadcasts with the classic broadcast `Intent` system.
- Ordered event processing as an option, akin to ordered broadcasts.
- Whereas Otto matches events only on the concrete base event class itself, greenrobot's EventBus also allows you to register event handlers for event class superclasses. For example, you could have a common `AppEvent` base class, with subclasses for specific scenarios, and have some object register

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

with an `onEvent(AppEvent)` method to find out about *all* of your events that inherit from `AppEvent`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Home Screen Effects

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Home Screen App Widgets

One of the oft-requested features added in Android 1.5 was the ability to add live elements to the home screen. Called "app widgets", these can be added by users via a long-tap on the home screen and choosing an appropriate widget from the available roster. Android ships with a few app widgets, such as a music player, but developers can add their own — in this chapter, we will see how this is done.

For the purposes of this book, "app widgets" will refer to these items that go on the home screen. Other uses of the term "widget" will be reserved for the UI widgets, subclasses of `View`, usually found in the `android.widget` Java package.

In this chapter, we briefly touch on the security ramifications of app widgets, before continuing on to discuss how Android offers a secure app widget framework. We then go through all the steps of creating a basic app widget. Next, we discuss how to deal with multiple instances of your app widget, the app widget lifecycle, alternative models for updating app widgets, and how to offer multiple layouts for your app widget (perhaps based on device characteristics). We wrap with some notes about hosting your own app widgets in your own home screen implementation.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapters on:

- basic widgets
- broadcast `Intents`
- services

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

East is East, and West is West…

Part of the reason it took as long as it did for app widgets to become available is security.

Android's security model is based heavily on Linux user, file, and process security. Each application is (normally) associated with a unique user ID. All of its files are owned by that user, and its process(es) run as that user. This prevents one application from modifying the files of another or otherwise injecting their own code into another running process.

In particular, the core Android team wanted to find a way that would allow app widgets to be displayed by the home screen application, yet have their content come from another application. It would be dangerous for the home screen to run arbitrary code itself or somehow allow its UI to be directly manipulated by another process.

The app widget architecture, therefore, is set up to keep the home screen application independent from any code that puts app widgets on that home screen, so bugs in one cannot harm the other.

The Big Picture for a Small App Widget

The way Android pulls off this bit of security is through the use of `RemoteViews`.

The application component that supplies the UI for an app widget is not an `Activity`, but rather a `BroadcastReceiver` (often in tandem with a `Service`). The `BroadcastReceiver`, in turn, does not inflate a normal `View` hierarchy, like an `Activity` would, but instead inflates a layout into a `RemoteViews` object.

`RemoteViews` encapsulates a limited edition of normal widgets, in such a fashion that the `RemoteViews` can be "easily" transported across process boundaries. You configure the `RemoteViews` via your `BroadcastReceiver` and make those `RemoteViews` available to Android. Android in turn delivers the `RemoteViews` to the app widget host (usually the home screen), which renders them to the screen itself.

This architectural choice has many impacts:

- You do not have access to the full range of widgets and containers. You can use `FrameLayout`, `LinearLayout`, and `RelativeLayout` for containers, and

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

AnalogClock, Button, Chronometer, ImageButton, ImageView, ProgressBar, and TextView for widgets. And, on API Level 11 and higher, you can use some AdapterView-based widgets, like ListView, as we will examine in the next chapter. And, as of API Level 16 (Android 4.1), you can use GridLayout... but not its backport on earlier devices.

- The only user input you can get is clicks of the Button and ImageButton widgets. In particular, there is no EditText for text input.
- Because the app widgets are rendered in another process, you cannot simply register an OnClickListener to get button clicks; rather, you tell RemoteViews a PendingIntent to invoke when a given button is clicked.
- You do not hold onto the RemoteViews and reuse them yourself. Rather, the pattern appears to be that you create and send out a brand-new RemoteViews whenever you want to change the contents of the app widget. This, coupled with having to transport the RemoteViews across process boundaries, means that updating the app widget is rather expensive in terms of CPU time, memory, and battery life.
- Because the component handling the updates is a BroadcastReceiver, you have to be quick (lest you take too long and Android consider you to have timed out), you cannot use background threads, and your component itself is lost once the request has been completed. Hence, if your update might take a while, you will probably want to have the BroadcastReceiver start a Service and have the Service do the long-running task and eventual app widget update.

Crafting App Widgets

This will become somewhat easier to understand in the context of some sample code. In the AppWidget/PairOfDice project, you will find an app widget that displays a roll of a pair of dice. Clicking on the app widget re-rolls, in case you want a better result.

The Manifest

First, we need to register our BroadcastReceiver implementation in our AndroidManifest.xml file, along with a few extra features:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.appwidget.dice"
  android:versionCode="1"
  android:versionName="1.0">
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"/>

  <uses-feature
    android:name="android.software.app_widgets"
    android:required="true"/>

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <receiver
      android:name=".AppWidget"
      android:icon="@drawable/cw"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE"/>
      </intent-filter>

      <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/widget_provider"/>
    </receiver>

    <activity
      android:name="PairOfDiceActivity"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Here, along with a do-nothing activity, we have a `<receiver>`. Of note:

1. Our `<receiver>` has `android:label` and `android:icon` attributes, which are not normally needed on `BroadcastReceiver` declarations. However, in this case, those are used for the entry that goes in the menu of available widgets to add to the home screen. Hence, you will probably want to supply values

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

for both of those, and use appropriate resources in case you want translations for other languages.

2. Our `<receiver>` has an `<intent-filter>` for the `android.appwidget.action.APPWIDGET_UPDATE` action. This means we will get control whenever Android wants us to update the content of our app widget. There may be other actions we want to monitor — more on this in a later section.
3. Our `<receiver>` also has a `<meta-data>` element, indicating that its `android.appwidget.provider` details can be found in the `res/xml/widget_provider.xml` file. This metadata is described in the next section.

The uses-feature Element

If the central point of your application is to provide an app widget, you should strongly consider adding a `<uses-feature>` element to advertise this fact to markets like the Play Store:

```
<uses-feature android:name="android.software.app_widgets"
android:required="true" />
```

In principle, having this element means that markets should block the installation of your app on devices where there is no app-widget-capable home screen or other known places for supporting app widgets.

If, however, your app has an app widget, but it is an adjunct to other forms of UI (typically a launcher activity), then you may wish to leave off this `<uses-feature>` element, or set it to `android:required="false"`.

The Metadata

Next, we need to define the app widget provider metadata. This has to reside at the location indicated in the manifest — in this case, in `res/xml/widget_provider.xml`:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="144dip"
  android:minHeight="72dip"
  android:updatePeriodMillis="900000"
  android:initialLayout="@layout/widget"
/>
```

Here, we provide four pieces of information:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. The minimum width and height of the app widget (`android:minWidth` and `android:minHeight`). These are approximate — the app widget host (e.g., home screen) will tend to convert these values into "cells" based upon the overall layout of the UI where the app widgets will reside. However, they should be no smaller than the minimums cited here. Also, ideally, you use `dip` instead of `px` for the dimensions, so the number of cells will remain constant regardless of screen density.
2. The frequency in which Android should request an update of the widget's contents (`android:updatePeriodMillis`). This is expressed in terms of milliseconds, so a value of `3600000` is a 60-minute update cycle. Note that the minimum value for this attribute is 30 minutes — values less than that will be "rounded up" to 30 minutes. Hence our 15-minute (`900000` millisecond) request will actually result in an update every 30 minutes.
3. The initial layout to use for the app widget, for the time between when the user requests the app widget and when `onUpdate()` of our `AppWidgetProvider` gets control.

Note that the calculations for determining the number of cells for an app widget varies. The `dip` dimension value for an N-cell dimension was (74 * N) - 2 (e.g., a 2x3 cell app widget would request a width of `146dip` and a height of `220dip`). The value as of API Level 14 (a.k.a., Ice Cream Sandwich) is now (70 * N) - 30 (e.g., a 2x3 cell app widget would request a width of `110dip` and a height of `180dip`). To have your app widgets maintain a consistent number of cells, you will need two versions of your app widget metadata XML, one in `res/xml-v14/` (with the API Level 14 calculation) and one in `res/xml/` (for prior versions of Android).

The Layout

Eventually, you are going to need a layout that describes what the app widget looks like. You need to stick to the widget and container classes noted above; otherwise, this layout works like any other layout in your project.

For example, here is the layout for the `PairOfDice` app widget:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/background"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/widget_frame"
    >
  <ImageView android:id="@+id/left_die"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:layout_centerVertical="true"
    android:layout_alignParentLeft="true"
    android:src="@drawable/die_5"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="7dip"
  />
  <ImageView android:id="@+id/right_die"
    android:layout_centerVertical="true"
    android:layout_alignParentRight="true"
    android:src="@drawable/die_2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginRight="7dip"
  />
</RelativeLayout>
```

All we have is a pair of `ImageView` widgets (one for each die), inside of a `RelativeLayout`. The `RelativeLayout` has a background, specified as a nine-patch PNG file. This allows the `RelativeLayout` to have guaranteed contrast with whatever wallpaper is behind it, so the user can tell the actual app widget bounds.

The BroadcastReceiver

Next, we need a `BroadcastReceiver` that can get control when Android wants us to update our `RemoteViews` for our app widget. To simplify this, Android supplies an `AppWidgetProvider` class we can extend, instead of the normal `BroadcastReceiver`. This simply looks at the received `Intent` and calls out to an appropriate lifecycle method based on the requested action.

The one method that invariably needs to be implemented on the provider is `onUpdate()`. Other lifecycle methods may be of interest and are discussed later in this chapter.

For example, here is the implementation of the `AppWidgetProvider` for `PairOfDice`:

```
package com.commonsware.android.appwidget.dice;

import android.app.PendingIntent;
import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.widget.RemoteViews;

public class AppWidget extends AppWidgetProvider {
  private static final int[] IMAGES={R.drawable.die_1,R.drawable.die_2,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                                    R.drawable.die_3,R.drawable.die_4,
                                    R.drawable.die_5,R.drawable.die_6};

  @Override
  public void onUpdate(Context ctxt, AppWidgetManager mgr,
                       int[] appWidgetIds) {
    ComponentName me=new ComponentName(ctxt, AppWidget.class);

    mgr.updateAppWidget(me, buildUpdate(ctxt, appWidgetIds));
  }

  private RemoteViews buildUpdate(Context ctxt, int[] appWidgetIds) {
    RemoteViews updateViews=new RemoteViews(ctxt.getPackageName(),
                                            R.layout.widget);

    Intent i=new Intent(ctxt, AppWidget.class);

    i.setAction(AppWidgetManager.ACTION_APPWIDGET_UPDATE);
    i.putExtra(AppWidgetManager.EXTRA_APPWIDGET_IDS, appWidgetIds);

    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0 , i,

PendingIntent.FLAG_UPDATE_CURRENT);

    updateViews.setImageViewResource(R.id.left_die,
                                     IMAGES[(int)(Math.random()*6)]);
    updateViews.setOnClickPendingIntent(R.id.left_die, pi);
    updateViews.setImageViewResource(R.id.right_die,
                                     IMAGES[(int)(Math.random()*6)]);
    updateViews.setOnClickPendingIntent(R.id.right_die, pi);
    updateViews.setOnClickPendingIntent(R.id.background, pi);

    return(updateViews);
  }
}
```

To update the `RemoteViews` for our app widget, we need to build those `RemoteViews` (delegated to a `buildUpdate()` helper method) and tell an `AppWidgetManager` to update the widget via `updateAppWidget()`. In this case, we use a version of `updateAppWidget()` that takes a `ComponentName` as the identifier of the widget to be updated. Note that this means that we will update all instances of this app widget presently in use — the concept of multiple app widget instances is covered in greater detail later in this chapter.

Working with `RemoteViews` is a bit like trying to tie your shoes while wearing mittens — it may be possible, but it is a bit clumsy. In this case, rather than using methods like `findViewById()` and then calling methods on individual widgets, we need to call methods on `RemoteViews` itself, providing the identifier of the widget we wish to modify. This is so our requests for changes can be serialized for transport to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the home screen process. It does, however, mean that our view-updating code looks a fair bit different than it would if this were the main `View` of an activity or row of a `ListView`.

To create the `RemoteViews`, we use a constructor that takes our package name and the identifier of our layout. This gives us a `RemoteViews` that contains all of the widgets we declared in that layout, just as if we inflated the layout using a `LayoutInflater`. The difference, of course, is that we have a `RemoteViews` object, not a `View`, as the result.

We then use methods like:

1. `setImageViewResource()` to set the image for each of our `ImageView` widgets, in this case a randomly chosen die face (using graphics created from a set of SVG files from the OpenClipArt site)
2. `setOnClickPendingIntent()` to provide a `PendingIntent` that should get fired off when a die, or the overall app widget background, is clicked

We then supply that `RemoteViews` to the `AppWidgetManager`, which pushes the `RemoteViews` structure to the home screen, which renders our new app widget UI.

The Result

If you compile and install all of this, you will have a new app widget entry available. How you add app widgets to the home screen varies based upon Android version and the home screen implementation, and there are too many possibilities to try to list here.

No matter how you add the Pair of Dice, the app widget will appear on the home screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

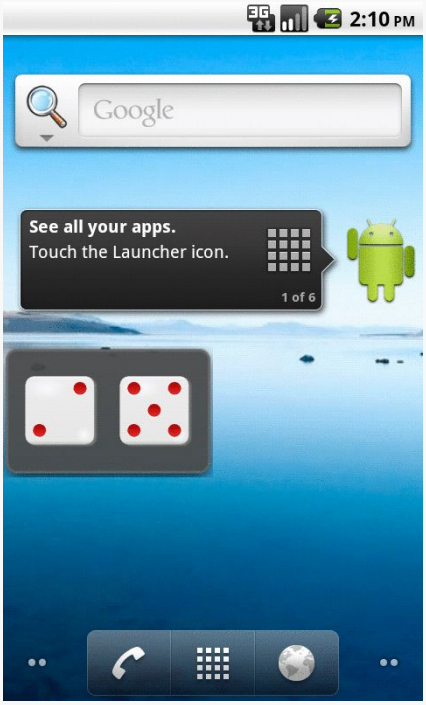


Figure 571: Pair of Dice, In Action

Another and Another

As indicated above, you can have multiple instances of the same app widget outstanding at any one time. For example, one might have multiple picture frames, or multiple "show-me-the-latest-RSS-entry" app widgets, one per feed. You will distinguish between these in your code via the identifier supplied in the relevant `AppWidgetProvider` callbacks (e.g., `onUpdate()`).

If you want to support separate app widget instances, you will need to store your state on a per-app-widget-identifier basis. You will also need to use an appropriate version of `updateAppWidget()` on `AppWidgetManager` when you update the app widgets, one that takes app widget identifiers as the first parameter, so you update the proper app widget instances.

Conversely, there is nothing requiring you to support multiple instances as independent entities. For example, if you add more than one `PairOfDice` app widget to your home screen, nothing blows up – they just show the same roll. That is because `PairOfDice` uses a version of `updateAppWidget()` that does not take any app widget IDs, and therefore updates all app widgets simultaneously.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

App Widgets: Their Life and Times

There are three other lifecycle methods that `AppWidgetProvider` offers that you may be interested in:

1. `onEnabled()` will be called when the first widget instance is created for this particular widget provider, so if there is anything you need to do once for all supported widgets, you can implement that logic here
2. `onDeleted()` will be called when a widget instance is removed from the home screen, in case there is any data you need to clean up specific to that instance
3. `onDisabled()` will be called when the last widget instance for this provider is removed from the home screen, so you can clean up anything related to all such widgets

Note, however, that there is a bug in Android 1.5, where `onDeleted()` will not be properly called. You will need to implement `onReceive()` and watch for the `ACTION_APPWIDGET_DELETED` action in the received `Intent` and call `onDeleted()` yourself. This has since been fixed, and if you are not supporting Android 1.5, you will not need to worry about this problem.

Controlling Your (App Widget's) Destiny

As `PairOfDice` illustrates, you are not limited to updating your app widget only based on the timetable specified in your metadata. That timetable is useful if you can get by with a fixed schedule. However, there are cases in which that will not work very well:

1. If you want the user to be able to configure the polling period (the metadata is baked into your APK and therefore cannot be modified at runtime)
2. If you want the app widget to be updated based on external factors, such as a change in location

The recipe shown in `PairOfDice` will let you use `AlarmManager` (described in another chapter) or proximity alerts or whatever to trigger updates. All you need to do is:

1. Arrange for something to broadcast an `Intent` that will be picked up by the `BroadcastReceiver` you are using for your app widget provider
2. Have the provider process that `Intent` directly or pass it along to a `Service` (such as an `IntentService`)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Also, note that the `updatePeriodMillis` setting not only tells the app widget to update every so often, it will even *wake up the phone* if it is asleep so the widget can perform its update. On the plus side, this means you can easily keep your widgets up to date regardless of the state of the device. On the minus side, this will tend to drain the battery, particularly if the period is too fast. If you want to avoid this wakeup behavior, set `updatePeriodMillis` to `0` and use `AlarmManager` to control the timing and behavior of your widget updates.

Note that if there are multiple instances of your app widget on the user's home screen, they will all update approximately simultaneously if you are using `updatePeriodMillis`. If you elect to set up your own update schedule, you can control which app widgets get updated when, if you choose.

Change Your Look

If you have been doing most of your development via the Android emulator, you are used to all "devices" having a common look and feel, in terms of the home screen, lock screen, and so forth. This is the so-called "Google Experience" look, and many actual Android devices have it.

However, some devices have their own presentation layers. HTC has "Sense", seen on the HTC Hero and HTC Tattoo, among other devices. Motorola has MOTOBLUR, seen on the Motorola CLIQ and DEXT. Other device manufacturers, like Sony Ericsson, Samsung, and LG, have followed suit, as will others in the future. These presentation layers replace the home screen and lock screen, among other things. Moreover, they usually come with their own suite of app widgets with their own look and feel. Your app widget may look fine on a Google Experience home screen, but the look might clash when viewed on a Sense or MOTOBLUR device.

Fortunately, there are ways around this. You can set your app widget's look on the fly at runtime, to choose the layout that will look the best on that particular device.

The first step is to create an app widget layout that is initially invisible (`res/layout/invisible.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:visibility="invisible"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    >
</RelativeLayout>
```

This layout is then the one you would reference from your app widget metadata, to be used when the app widget is first created:

```
<appwidget-provider
xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="292dip"
  android:minHeight="72dip"
  android:updatePeriodMillis="900000"
  android:configure="com.commonsware.android.appwidget.TWPrefs"
  android:initialLayout="@layout/invisible"
/>
```

This ensures that when your app widget is initially added, you do not get the "Problem loading widget" placeholder, yet you also do not choose one layout versus another — it is simply invisible for a brief moment.

Then, in your `AppWidgetProvider` (or attached `IntentService`), you can make the choice of what layout to inflate as part of your `RemoteViews`. Rather than using the invisible one, you can choose one based on the device or other characteristics. The biggest challenge is that there is no good way to determine what presentation layer, if any, is in use on a device. For the time being, you will need to use the various fields in the `android.os.Build` class to "sniff" on the device model and make a decision that way.

One Size May Not Fit All

It may be that you want to offer multiple app widget sizes to your users. Some might only want a small app widget. Some might really like what you have to offer and want to give you more home screen space to work in.

Android 1.x/2.x

The good news: this is easy to do.

The bad news: it requires you, in effect, to have one app widget per size.

The size of an app widget is determined by the app widget metadata XML file. That XML file is tied to a `<receiver>` element in the manifest representing one app widget. Hence, to have multiple sizes, you need multiple metadata files and multiple `<receiver>` elements.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This also means your app widgets will show up multiple times in the app widget selection list, when the user goes to add an app widget to their home screen. Hence, supporting many sizes will become annoying to the user, if they perceive you are "spamming" the app widget list. Try to keep the number of app widget sizes to a reasonable number (say, one or two sizes).

Android 3.0+

As of API Level 11, it is possible to have a resizeable app widget. To do this, you can have an `android:resizeMode` attribute in your widget metadata, with a value of `horizontal`, `vertical`, or both (e.g., `horizontal|vertical`). When the user long-taps on an existing widget, they should see handles to allow the widget to be resized:

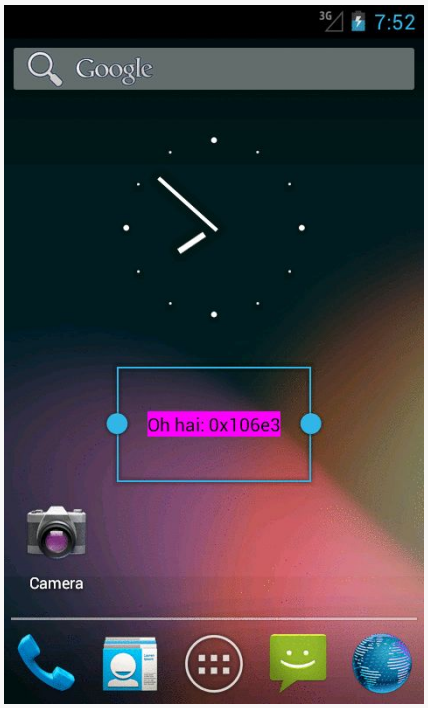


Figure 572: API Demos App Widget, Resizing

You can also have `android:minResizeWidth` and `android:minResizeHeight` attributes, measured in `dp`, that indicate the approximate smallest size that your app widget can support. These values will be interpreted in terms of "cells", as with the `android:minWidth` and `android:minHeight` attributes, and so the `dp` values you supply will not be used precisely.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, for Android 3.x and 4.0 (API Level 11–15), your code would not be informed about being resized. You had to simply ensure that your layout would intelligently use any extra space automatically. Hence, resizing tended to be used primarily with adapter-driven app widgets, as will be discussed in the next chapter.

Starting with API Level 16, though, you can find out when the user resizes your app widget, so you can perhaps use a different layout for a different size, or otherwise adapt to the available space. Finding out about resize events takes a bit more work, as is illustrated in the AppWidget/Resize sample project.

This app widget project is similar to `PairOfDice`, described earlier in this chapter. However, our layout skips the dice, replacing them with a `TextView` widget in the `RelativeLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/background"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="@drawable/widget_frame"
  android:orientation="horizontal">

  <TextView
    android:id="@+id/size"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:textAppearance="?android:attr/textAppearanceMedium">

  </TextView>

</RelativeLayout>
```

Our `widget_provider.xml` resource stipulates our desired `android:resizeMode` and minimum resize dimensions:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="180dip"
  android:minHeight="110dip"
  android:minResizeWidth="110dip"
  android:minResizeHeight="40dip"
  android:initialLayout="@layout/widget"
  android:resizeMode="horizontal|vertical"
/>
```

Finding out about app widget resizing is a different event than finding out about app widget updates. Hence, we need to add a new `<action>` element to the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

<intent-filter> of our <receiver> in the manifest, indicating that we want APPWIDGET_OPTIONS_CHANGED as well as ACTION_UPDATE:

```
  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <receiver
      android:name="AppWidget"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE"/>
        <action
android:name="android.appwidget.action.APPWIDGET_OPTIONS_CHANGED"/>
      </intent-filter>
```

Then, our app widget implementation can override an onAppWidgetOptionsChanged() method:

```
  @Override
  public void onAppWidgetOptionsChanged(Context ctxt,
                                        AppWidgetManager mgr,
                                        int appWidgetId,
                                        Bundle newOptions) {
    RemoteViews updateViews=
        new RemoteViews(ctxt.getPackageName(), R.layout.widget);
    String msg=
        String.format(Locale.getDefault(),
                      "[%d-%d] x [%d-%d]",

newOptions.getInt(AppWidgetManager.OPTION_APPWIDGET_MIN_WIDTH),

newOptions.getInt(AppWidgetManager.OPTION_APPWIDGET_MAX_WIDTH),

newOptions.getInt(AppWidgetManager.OPTION_APPWIDGET_MIN_HEIGHT),

newOptions.getInt(AppWidgetManager.OPTION_APPWIDGET_MAX_HEIGHT));

    updateViews.setTextViewText(R.id.size, msg);

    mgr.updateAppWidget(appWidgetId, updateViews);
  }
```

You will notice that we skip onUpdate(). We will be called with onAppWidgetOptionsChanged() when the app widget is added and resized. Hence, in the case of this app widget, we can define what the app widget looks like from onAppWidgetOptionsChanged(), eschewing onUpdate(). That being said, more typical app widgets will wind up implementing both methods, especially if they are

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

supporting lower API levels than 16, where `onAppWidgetOptionsChanged()` will not be called.

Also remember that your process may well be terminated in between calls to app widget lifecycle methods like `onUpdate()` and `onAppWidgetOptionsChanged()`. Hence, if there is data from one method that you want in the other, be sure to persist that data somewhere.

In the `AppWidget` implementation of `onAppWidgetOptionsChanged()`, we can find out about our new app widget size by means of the `Bundle` supplied to our method. What we *cannot* find out is our exact size. Rather, we are provided minimum and maximum dimensions of our app widget via four values in the `Bundle`:

- `AppWidgetManager.OPTION_APPWIDGET_MIN_WIDTH`
- `AppWidgetManager.OPTION_APPWIDGET_MAX_WIDTH`
- `AppWidgetManager.OPTION_APPWIDGET_MIN_HEIGHT`
- `AppWidgetManager.OPTION_APPWIDGET_MAX_HEIGHT`

In our case, we grab these `int` values and pour them into a `String` template, using that to fill in the `TextView` of the app widget's contents.

When our app widget is initially launched, we show our initial size ranges:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

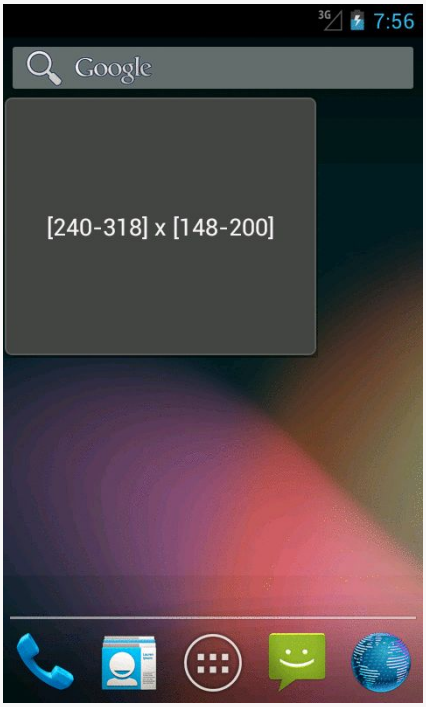


Figure 573: Resize Widget, As Initially Added

When the user resizes our app widget, we show the new size ranges:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

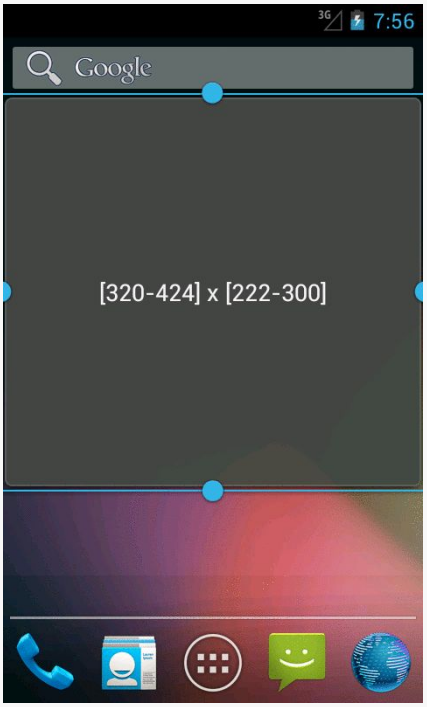


Figure 574: Resize Widget, During Resize Operation

However, not all home screen implementations will necessarily send the `APPWIDGET_OPTIONS_CHANGED` when an app widget is added to the home screen, only when the user resizes it later. For example, while the emulator's home screen for Android 4.1 broadcasts `APPWIDGET_OPTIONS_CHANGED`, it does not for 4.2 or 4.3. Hence, you may want to also examine the size information in `onUpdate()` as well, so that you react to the initial size as well as any future sizes. One way to do this is to simply iterate over the supplied app widget IDs and invoke your own `onAppWidgetOptionsChanged()` method:

```
// based on http://stackoverflow.com/a/18552461/115145

@Override
public void onUpdate(Context context,
                     AppWidgetManager appWidgetManager,
                     int[] appWidgetIds) {
  super.onUpdate(context, appWidgetManager, appWidgetIds);

  for (int appWidgetId : appWidgetIds) {
    Bundle options=appWidgetManager.getAppWidgetOptions(appWidgetId);

    onAppWidgetOptionsChanged(context, appWidgetManager, appWidgetId,
                              options);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }
```

Lockscreen Widgets

Android's lockscreen (a.k.a., the keyguard) has long been unmodifiable by developers. This led to a number of developers creating so-called "replacement lockscreens", which generally reduce device security, as they can be readily bypassed. However, with API Level 17 (Android 4.2), developers now can create app widgets that the user can deploy to the lockscreen, helping to eliminate the need for "replacement lockscreens".

Declaring that an app widget supports being on the lockscreen instead of (or in addition to) the home screen is very easy. All you must do is add an `android:widgetCategory` attribute to your app widget metadata resource. That attribute should have a value of either `keyguard` (for the lockscreen), `home_screen`, or both (e.g., `keyguard|home_screen`), depending upon where you want the app widget to be eligible. By default, if this attribute is missing, Android assumes a default value of `home_screen`.

Users cannot resize the lockscreen widgets at this time. However, you still will want to specify an `android:resizeMode` attribute in your app widget metadata, as whether or not you include `vertical` resizing will affect the height of your app widget. Lockscreen widgets without `vertical` will have a fixed small height on tablets, while lockscreen widgets with `vertical` will fill the available height. Lockscreen widgets on phones will always be small (to fit above the PIN/password entry area), and lockscreen widgets on all devices will stretch to fill available space horizontally.

You can also specify a different starting layout to use when your app is added to the lockscreen, as opposed to being added to the home screen. To do this, just add an `android:initialKeyguardLayout` attribute to your app widget metadata, pointing to the lockscreen-specific layout to use.

To see this in action, take a look at the `AppWidget/TwoOrThreeDice` sample project. This is a revised clone of the `PairOfDice` sample, allowing the dice to be added to the lockscreen, and showing three dice on the lockscreen instead of the two on the home screen.

Our app widget metadata now contains the lockscreen-related attributes: `android:widgetCategory` and `android:initialKeyguardLayout`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="144dip"
  android:minHeight="72dip"
  android:updatePeriodMillis="900000"
  android:initialLayout="@layout/widget"
  android:initialKeyguardLayout="@layout/lockscreen"
  android:widgetCategory="keyguard|home_screen"
/>
```

Our `lockscreen` layout simply adds a third die, `middle_die`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/background"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/widget_frame"
    >
  <ImageView android:id="@+id/left_die"
    android:layout_centerVertical="true"
    android:layout_alignParentLeft="true"
    android:src="@drawable/die_3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="7dip"
  />
  <ImageView android:id="@+id/middle_die"
    android:layout_centerInParent="true"
    android:src="@drawable/die_2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="7dip"
    android:layout_marginRight="7dip"
  />
  <ImageView android:id="@+id/right_die"
    android:layout_centerVertical="true"
    android:layout_alignParentRight="true"
    android:src="@drawable/die_2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginRight="7dip"
  />
</RelativeLayout>
```

However, by specifying a different layout for the lockscreen widget, we have a problem. We need to know, in our Java code, what layout to use for the `RemoteViews` and how many dice need to be updated. And, ideally, we would handle this in a backwards-compatible fashion, so our app widget will have its original functionality on older Android devices. Plus, supporting the lockscreen makes it that much more likely that the user will have more than one instance of our app widget (e.g., one on

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the lockscreen and one on the homescreen), so we should do a better job than `PairOfDice` did about handling multiple app widget instances.

To deal with the latter point, our new `onUpdate()` method iterates over each of the app widget IDs supplied to it and calls a private `updateWidget()` method for each, so we can better support multiple instances:

```
@Override
public void onUpdate(Context ctxt, AppWidgetManager mgr,
                     int[] appWidgetIds) {
  for (int appWidgetId : appWidgetIds) {
    updateWidget(ctxt, mgr, appWidgetId);
  }
}
```

The `updateWidget()` method is a bit more complicated than the `PairOfDice` equivalent code:

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
private void updateWidget(Context ctxt, AppWidgetManager mgr,
                          int appWidgetId) {
  int layout=R.layout.widget;

  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
    int category=
        mgr.getAppWidgetOptions(appWidgetId)
           .getInt(AppWidgetManager.OPTION_APPWIDGET_HOST_CATEGORY,
                   -1);

    layout=
        (category == AppWidgetProviderInfo.WIDGET_CATEGORY_KEYGUARD)
            ? R.layout.lockscreen : R.layout.widget;
  }

  RemoteViews updateViews=
      new RemoteViews(ctxt.getPackageName(), layout);
  Intent i=new Intent(ctxt, AppWidget.class);

  i.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);

  PendingIntent pi=
      PendingIntent.getBroadcast(ctxt, appWidgetId, i,
                                 PendingIntent.FLAG_UPDATE_CURRENT);

  updateViews.setImageViewResource(R.id.left_die,
                                   IMAGES[(int)(Math.random() * 6)]);
  updateViews.setOnClickPendingIntent(R.id.left_die, pi);
  updateViews.setImageViewResource(R.id.right_die,
                                   IMAGES[(int)(Math.random() * 6)]);
  updateViews.setOnClickPendingIntent(R.id.right_die, pi);
  updateViews.setOnClickPendingIntent(R.id.background, pi);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (layout == R.layout.lockscreen) {
      updateViews.setImageViewResource(R.id.middle_die,
                                       IMAGES[(int)(Math.random() * 6)]);
      updateViews.setOnClickPendingIntent(R.id.middle_die, pi);
    }

    mgr.updateAppWidget(appWidgetId, updateViews);
  }
```

First, we need to choose which layout we are working with. We assume that we are to use the original `R.layout.widget` resource by default. But, if we are on API Level 17 or higher, we can call `getAppWidgetOptions()` on the `AppWidgetManager`, to get the `Bundle` of options — the same options that we could be delivered in `onAppWidgetOptionsUpdate()` as described in the previous section. One value that will be in this `Bundle` is `AppWidgetManager.OPTION_APPWIDGET_HOST_CATEGORY`, which will be an `int` with a value of `AppWidgetProviderInfo.WIDGET_CATEGORY_KEYGUARD` if our app widget is on the lockscreen. In that case, we switch to using `R.layout.lockscreen`. In addition, we know then we need to update the `middle_die` when we are updating the other dice.

There is also a subtle change in our `getBroadcast()` call to `PendingIntent`: we pass in the app widget ID as the second parameter, whereas in `PairOfDice` we passed `0`. `PendingIntent` objects are cached in our process, and by default we will get the same `PendingIntent` when we call `getBroadcast()` for the same `Intent`. However, in our case, we may *want* two or more different `PendingIntent` objects for the same `Intent`, with differing extras (`EXTRA_APPWIDGET_ID`). Since extras are not considered when evaluating equivalence of `Intent` objects, just having different extras is insufficient to get different `PendingIntent` objects for those `Intent` objects. The second parameter to `getBroadcast()` (and `getActivity()` and `getService()`) on `PendingIntent` is a unique identifier, to differentiate between two otherwise equivalent `Intent` objects, forcing `PendingIntent` to give us distinct `PendingIntent` objects. This way, we can support two or more app widget instances, each having their own `PendingIntent` objects for their click events.

On an Android 4.2 lockscreen, you should be able to swipe to one side (e.g., a bezel swipe from left to right), to expose an option to add an app widget:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

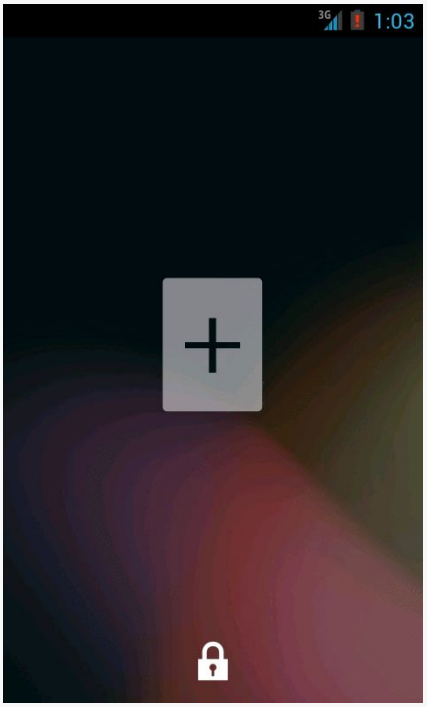


Figure 575: Lockscreen Add-A-Widget Panel, On a 4.2 Emulator

Tapping the "+" indicator (and, if needed, entering your device PIN or password), brings up an app widget chooser:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

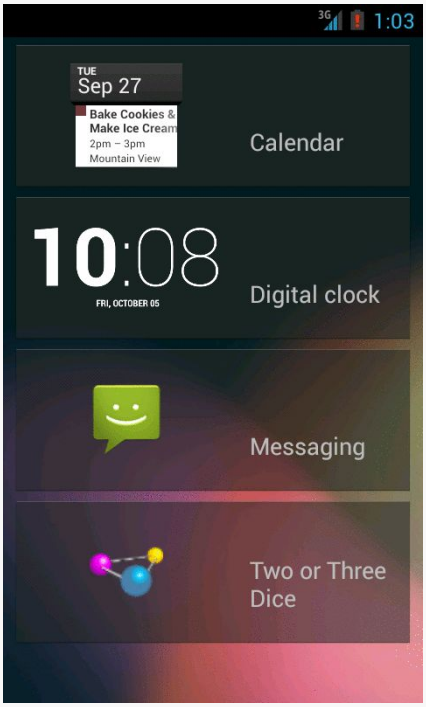


Figure 576: Lockscreen Widget Selection List, On a 4.2 Emulator

Choosing `TwoOrThreeDice` will then add the app widget to the lockscreen, with three dice, not two:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

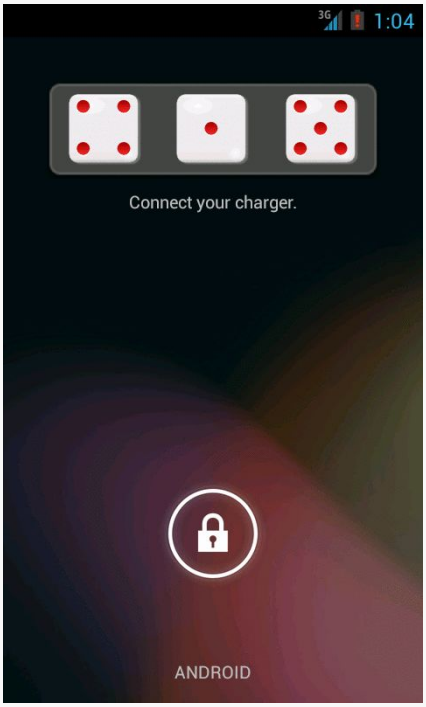


Figure 577: Lockscreen with TwoOrThreeDice, On a 4.2 Emulator

Preview Images

App widgets can now have preview images attached. Preview images are drawable resources representing a preview of what the app widget might look like on the screen. On tablets, this will be used as part of an app widget gallery, replacing the simple context menu presentation you see on Android 1.x and 2.x phones:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

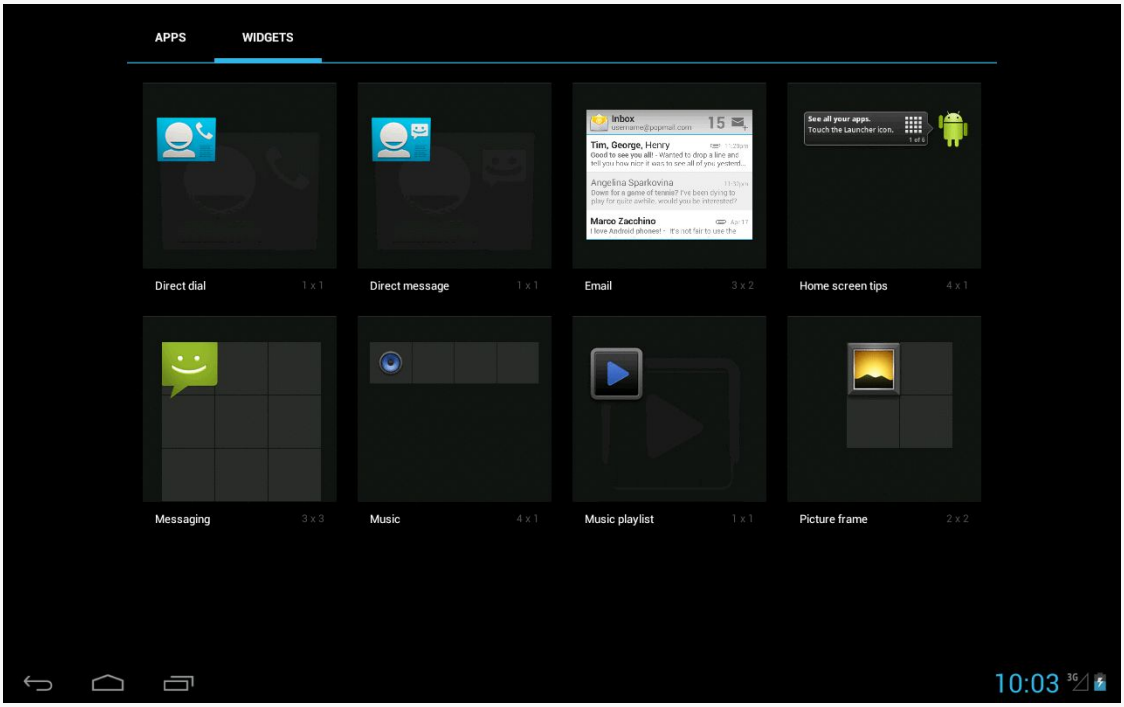


Figure 578: The XOOM tablet's app widget gallery

To create the preview image itself, the Android 3.0 emulator contains a Widget Preview application that lets you run an app widget in its own container, outside of the home screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

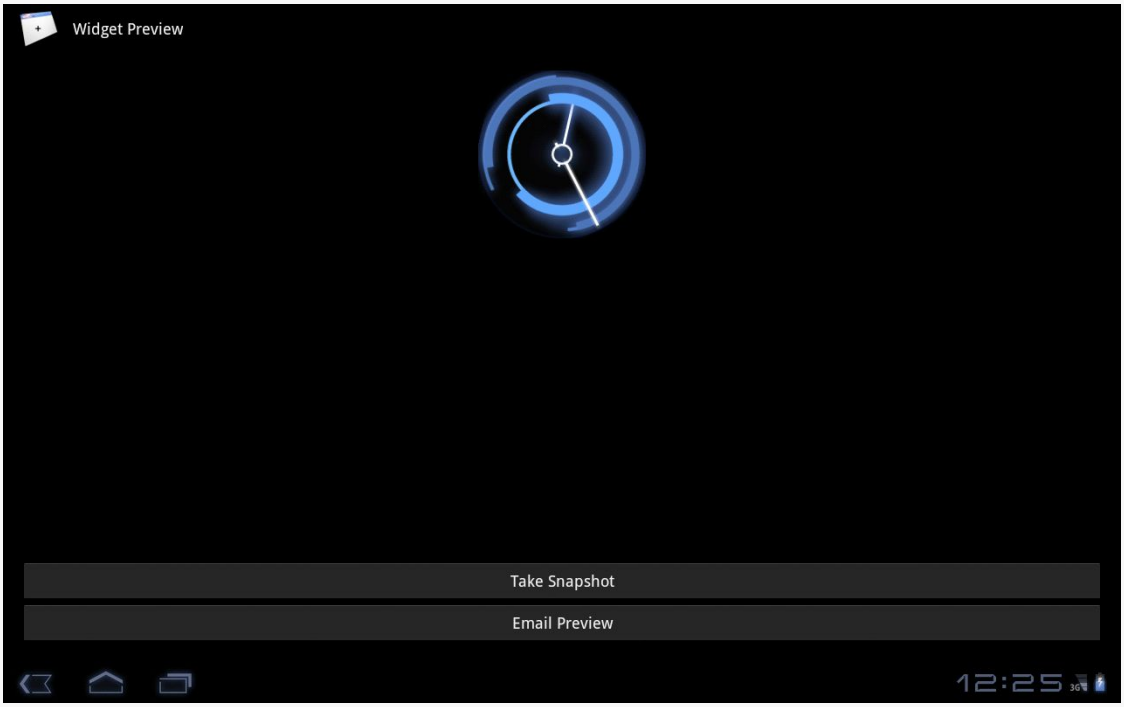


Figure 579: The Widget Preview application, showing a preview of the Analog Clock app widget

From here, you can take a snapshot and save it to external storage, copy it to your project's `res/drawable-nodpi/` directory (indicating that there is no intrinsic density assumed for this image), and reference it in your app widget metadata via an `android:previewImage` attribute. We will see an example of such an attribute in the chapter on advanced app widgets.

Being a Good Host

In addition to creating your own app widgets, it is possible to host app widgets. This is mostly aimed for those creating alternative home screen applications, so they can take advantage of the same app widget framework and all the app widgets being built for it.

This is not very well documented at this juncture, but it apparently involves the `AppWidgetHost` and `AppWidgetHostView` classes. The latter is a `View` and so should be able to reside in an app widget host's UI like any other ordinary widget.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Adapter-Based App Widgets

API Level 11 introduced a few new capabilities for app widgets, to make them more interactive and more powerful than before. The documentation lags a bit, though, so determining how to use these features takes a bit of exploring. Fortunately for you, the author did some of that exploring on your behalf, to save you some trouble.

Prerequisites

Understanding this chapter requires that you have read the preceding chapter and all of its prerequisites.

AdapterViews for App Widgets

In addition to the classic widgets available for use in app widgets and `RemoteViews`, five more were added for API Level 11:

1. `GridView`
2. `ListView`
3. `StackView`
4. `ViewFlipper`
5. `AdapterViewFlipper`

Three of these (`GridView`, `ListView`, `ViewFlipper`) are widgets that existed in Android since the outset. `StackView` is a new widget to provide a "stack of cards" UI:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 580: The Google Books app widget, showing a StackView

`AdapterViewFlipper` works like a `ViewFlipper`, allowing you to toggle between various children with only one visible at a time. However, whereas with `ViewFlipper` all children are fully-instantiated `View` objects held by the `ViewFlipper` parent, `AdapterViewFlipper` uses the `Adapter` model, so only a small number of actual `View` objects are held in memory, no matter how many potential children there are.

With the exception of `ViewFlipper`, the other four all require the use of an `Adapter`. This might seem odd, as there is no way to provide an `Adapter` to a `RemoteViews`. That is true, but Android 3.0 added new ways for `Adapter`-like communication between the app widget host (e.g., home screen) and your application. We will take an in-depth look at that in an upcoming section.

Building Adapter-Based App Widgets

In an activity, if you put a `ListView` or `GridView` into your layout, you will also need to hand it an `Adapter`, providing the actual row or cell `View` objects that make up the contents of those selection widgets.

In an app widget, this becomes a bit more complicated. The host of the app widget does not have any `Adapter` class of yours. Hence, just as we have to send the contents of the app widget's UI via a `RemoteViews`, we will need to provide the rows or cells via `RemoteViews` as well. Android, starting with API Level 11, has a `RemoteViewsService` and `RemoteViewsFactory` that you can use for this purpose.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Let's take a look, in the form of the AppWidget/LoremWidget sample project, which will put a ListView of 25 Latin words into an app widget.

The AppWidgetProvider

At its core, our AppWidgetProvider (named WidgetProvider, in a stunning display of creativity) still needs to create and configure a RemoteViews object with the app widget UI, then use updateAppWidget() to push that RemoteViews to the host via the AppWidgetManager. However, for an app widget that involves an AdapterView, like ListView, there are two more key steps:

- You have to tell the RemoteViews the identity of a RemoteViewsService that will help fill the role that the Adapter would in an activity
- You have to provide the RemoteViews with a "template" PendingIntent to be used when the user taps on a row or cell in the AdapterView, to replace the onListItemClick() or similar method you might have used in an activity

For example, here is WidgetProvider for our nonsense-word app widget:

```
package com.commonsware.android.appwidget.lorem;

import android.app.PendingIntent;
import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.Context;
import android.content.Intent;
import android.net.Uri;
import android.widget.RemoteViews;

public class WidgetProvider extends AppWidgetProvider {
  public static String EXTRA_WORD=
    "com.commonsware.android.appwidget.lorem.WORD";

  @Override
  public void onUpdate(Context ctxt, AppWidgetManager appWidgetManager,
                       int[] appWidgetIds) {
    for (int i=0; i<appWidgetIds.length; i++) {
      Intent svcIntent=new Intent(ctxt, WidgetService.class);

      svcIntent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetIds[i]);
      svcIntent.setData(Uri.parse(svcIntent.toUri(Intent.URI_INTENT_SCHEME)));

      RemoteViews widget=new RemoteViews(ctxt.getPackageName(),
                                         R.layout.widget);

      widget.setRemoteAdapter(R.id.words, svcIntent);

      Intent clickIntent=new Intent(ctxt, LoremActivity.class);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      PendingIntent clickPI=PendingIntent
                              .getActivity(ctxt, 0,
                                           clickIntent,
                                           PendingIntent.FLAG_UPDATE_CURRENT);

      widget.setPendingIntentTemplate(R.id.words, clickPI);

      appWidgetManager.updateAppWidget(appWidgetIds[i], widget);
    }

    super.onUpdate(ctxt, appWidgetManager, appWidgetIds);
  }
}
```

The call to `setRemoteAdapter()` is where we point the `RemoteViews` to our `RemoteViewsService` for our `AdapterView` widget. The main rules for the `Intent` used to identify the `RemoteViewsService` are:

1. The service must be identified by its data (`Uri`), so even if you create the `Intent` via the `Context`-and-`Class` constructor, you will need to convert that into a `Uri` via `toUri(Intent.URI_INTENT_SCHEME)` and set that as the `Uri` for the `Intent`. Why? While your application has access to your `RemoteViewsService` `Class` object, the app widget host will not, and so we need something that will work across process boundaries. You could elect to add your own `<intent-filter>` to the `RemoteViewsService` and use an `Intent` based on that, but that would make your service more publicly visible than you might want.
2. Any extras that you package on the `Intent` — such as the app widget ID in this case — will be on the `Intent` that is delivered to the `RemoteViewsService` when it is invoked by the app widget host.

Note that there are two flavors of `setRemoteAdapter()`. An older deprecated one takes the app widget ID as the first parameter. The current one does not. The current one, though, is only available on API Level 14 and higher.

The call to `setPendingIntentTemplate()` is where we provide a `PendingIntent` that will be used as the template for all row or cell clicks. As we will see in a bit, the underlying `Intent` in the `PendingIntent` will have more data added to it by our `RemoteViewsFactory`.

In all other respects, our `WidgetProvider` is unremarkable compared to other app widgets. It will need to be registered in the manifest as a `<receiver>`, as with any other app widget.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The RemoteViewsService

Android supplies a `RemoteViewsService` class that you will need to extend, and this class is the one you must register with the `RemoteViews` for an `AdapterView` widget. For example, here is `WidgetService` (once again, a highly creative name) from the `LoremWidget` project:

```
package com.commonsware.android.appwidget.lorem;

import android.content.Intent;
import android.widget.RemoteViewsService;

public class WidgetService extends RemoteViewsService {
  @Override
  public RemoteViewsFactory onGetViewFactory(Intent intent) {
    return(new LoremViewsFactory(this.getApplicationContext(),
                                 intent));
  }
}
```

As you can see, this service is practically trivial. You have to override one method, `onGetViewFactory()`, which will return the `RemoteViewsFactory` to use for supplying rows or cells for the `AdapterView`. You are passed in an `Intent`, the one used in the `setRemoteAdapter()` call. Hence, if you have more than one `AdapterView` widget in your app widget, you could elect to have two `RemoteViewsService` implementations, or one that discriminates between the two widgets via something in the `Intent` (e.g., custom action string). In our case, we only have one `AdapterView`, so we create an instance of a `LoremViewFactory` and return it. Google demonstrates using `getApplicationContext()` here to supply the `Context` object to `RemoteViewsFactory`, instead of using the Service as a `Context` — it is unclear at this time why this is.

Another thing different about the `RemoteViewsService` is how it is registered in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.appwidget.lorem"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="19"/>

  <uses-feature
    android:name="android.software.app_widgets"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:required="true"/>

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="LoremActivity"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>

    <receiver
      android:name="WidgetProvider"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE"/>
      </intent-filter>

      <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/widget_provider"/>
    </receiver>

    <service
      android:name="WidgetService"
      android:permission="android.permission.BIND_REMOTEVIEWS"/>
  </application>

</manifest>
```

Note the use of `android:permission`, specifying that whoever sends an `Intent` to `WidgetService` must hold the `BIND_REMOTEVIEWS` permission. This can only be held by the operating system. This is a security measure, so arbitrary applications cannot find out about your service and attempt to spoof being the OS and cause you to supply them with `RemoteViews` for the rows, as this might leak private data.

The RemoteViewsFactory

A `RemoteViewsFactory` interface implementation looks and feels a lot like an `Adapter`. In fact, one could imagine that the Android developer community might create `CursorRemoteViewsFactory` and `ArrayRemoteViewsFactory` and such to further simplify writing these classes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For example, here is `LoremViewsFactory`, the one used by the `LoremWidget` project:

```
package com.commonsware.android.appwidget.lorem;

import android.appwidget.AppWidgetManager;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.widget.RemoteViews;
import android.widget.RemoteViewsService;

public class LoremViewsFactory implements
    RemoteViewsService.RemoteViewsFactory {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };
  private Context ctxt=null;
  private int appWidgetId;

  public LoremViewsFactory(Context ctxt, Intent intent) {
    this.ctxt=ctxt;
    appWidgetId=
        intent.getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
                           AppWidgetManager.INVALID_APPWIDGET_ID);
  }

  @Override
  public void onCreate() {
    // no-op
  }

  @Override
  public void onDestroy() {
    // no-op
  }

  @Override
  public int getCount() {
    return(items.length);
  }

  @Override
  public RemoteViews getViewAt(int position) {
    RemoteViews row=
        new RemoteViews(ctxt.getPackageName(), R.layout.row);

    row.setTextViewText(android.R.id.text1, items[position]);

    Intent i=new Intent();
    Bundle extras=new Bundle();

    extras.putString(WidgetProvider.EXTRA_WORD, items[position]);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    extras.putInt(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);
    i.putExtras(extras);
    row.setOnClickFillInIntent(android.R.id.text1, i);

    return(row);
  }

  @Override
  public RemoteViews getLoadingView() {
    return(null);
  }

  @Override
  public int getViewTypeCount() {
    return(1);
  }

  @Override
  public long getItemId(int position) {
    return(position);
  }

  @Override
  public boolean hasStableIds() {
    return(true);
  }

  @Override
  public void onDataSetChanged() {
    // no-op
  }
}
```

You need to implement a handful of methods that have the same roles in a `RemoteViewsFactory` as they do in an `Adapter`, including:

1. `getCount()`
2. `getViewTypeCount()`
3. `getItemId()`
4. `hasStableIds()`

In addition, you have `onCreate()` and `onDestroy()` methods that you must implement, even if they do nothing, to satisfy the interface.

You will need to implement `getLoadingView()`, which will return a `RemoteViews` to use as a placeholder while the app widget host is getting the real contents for the app widget. If you return `null`, Android will use a default placeholder.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The bulk of your work will go in `getViewAt()`. This serves the same role as `getView()` does for an `Adapter`, in that it returns the row or cell `View` for a given position in your data set. However:

1. You have to return a `RemoteViews`, instead of a `View`, just as you have to use `RemoteViews` for the main content of the app widget in your `AppWidgetProvider`
2. There is no recycling, so you do not get a `View` (or `RemoteViews`) back to somehow repopulate, meaning you will create a new `RemoteViews` every time

The impact of the latter is that you do not want to put large data sets into an app widget, as scrolling may get sluggish, just as you do not want to implement an `Adapter` without recycling unused `View` objects.

In `LoremViewsFactory`, the `getViewAt()` implementation creates a `RemoteViews` for a custom row layout, cribbed from one in the Android SDK:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2006 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->

<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:gravity="center_vertical"
    android:paddingLeft="6dip"
    android:minHeight="?android:attr/listPreferredItemHeight"
/>
```

Then, `getViewAt()` pours in a word from the static `String` array of Latin words into that `RemoteViews` for the `TextView` inside it. It also creates an `Intent` and puts the Latin word in as an `EXTRA_WORD` extra, then provides that `Intent` to `setOnClickFillInIntent()`. In addition, it adds the app widget instance ID as an

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

extra, reusing the framework's own `AppWidgetManager.EXTRA_APPWIDGET_ID` as the key. The contents of the "fill-in" `Intent` are merged into the "template" `PendingIntent` from `setPendingIntentTemplate()`, and the resulting `PendingIntent` is what is invoked when the user taps on an item in the `AdapterView`. The fully-configured `RemoteViews` is then returned.

The Rest of the Story

The app widget metadata needs no changes related to `Adapter`-based app widget contents. However, `LoremWidget` does add the `android:previewImage` attribute:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="146dip"
  android:minHeight="146dip"
  android:updatePeriodMillis="0"
  android:initialLayout="@layout/widget"
  android:autoAdvanceViewId="@+id/words"
  android:previewImage="@drawable/preview"
  android:resizeMode="vertical"
/>
```

This points to the `res/drawable-nodpi/preview.png` file that represents a "widgetshot" of the app widget in isolation, obtained from the Widget Preview application:

Figure 581: The preview of LoremWidget

Also, the metadata specifies `android:resizeMode="vertical"`. This attribute is new to Android 3.1, and allows the app widget to be resized by the user (in this case, only in the vertical direction, to show more rows). Older versions of Android will ignore this attribute, and the app widget will remain in your requested size. You can use

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`vertical`, `horizontal`, or both (via the pipe operator) as values for `android:resizeMode`.

When the user taps on an item in the list, our `PendingIntent` is set to bring up `LoremActivity`. This activity has `android:theme="@android:style/Theme.NoDisplay"` set in the manifest, meaning that it will not have its own user interface. Rather, it will extract our `EXTRA_WORD` — and the app widget ID — out of the `Intent` used to launch the activity and displays them in a `Toast` before finishing:

```
package com.commonsware.android.appwidget.lorem;

import android.app.Activity;
import android.appwidget.AppWidgetManager;
import android.os.Bundle;
import android.widget.Toast;

public class LoremActivity extends Activity {
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    String word=getIntent().getStringExtra(WidgetProvider.EXTRA_WORD);

    if (word == null) {
      word="We did not get a word!";
    }

    Toast.makeText(this,
                   String.format("#%d: %s",

getIntent().getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,

AppWidgetManager.INVALID_APPWIDGET_ID),
                                   word), Toast.LENGTH_LONG).show();

    finish();
  }
}
```

The Results

When you compile and install the application, nothing new shows up in the home screen launcher, because we have no activity defined to respond to `ACTION_MAIN` and `CATEGORY_HOME`. This would be unusual for an application distributed through the Play Store, as users often get confused if they install something and then do not know how to start it. However, for the purposes of this example, we should be fine, as readers of programming books never get confused about such things.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, if you bring up the app widget gallery (e.g., long-tap on the home screen of a Motorola XOOM), you will see `LoremWidget` there, complete with preview image. You can drag it into one of the home screen panes and position it. When done, the app widget appears as expected:

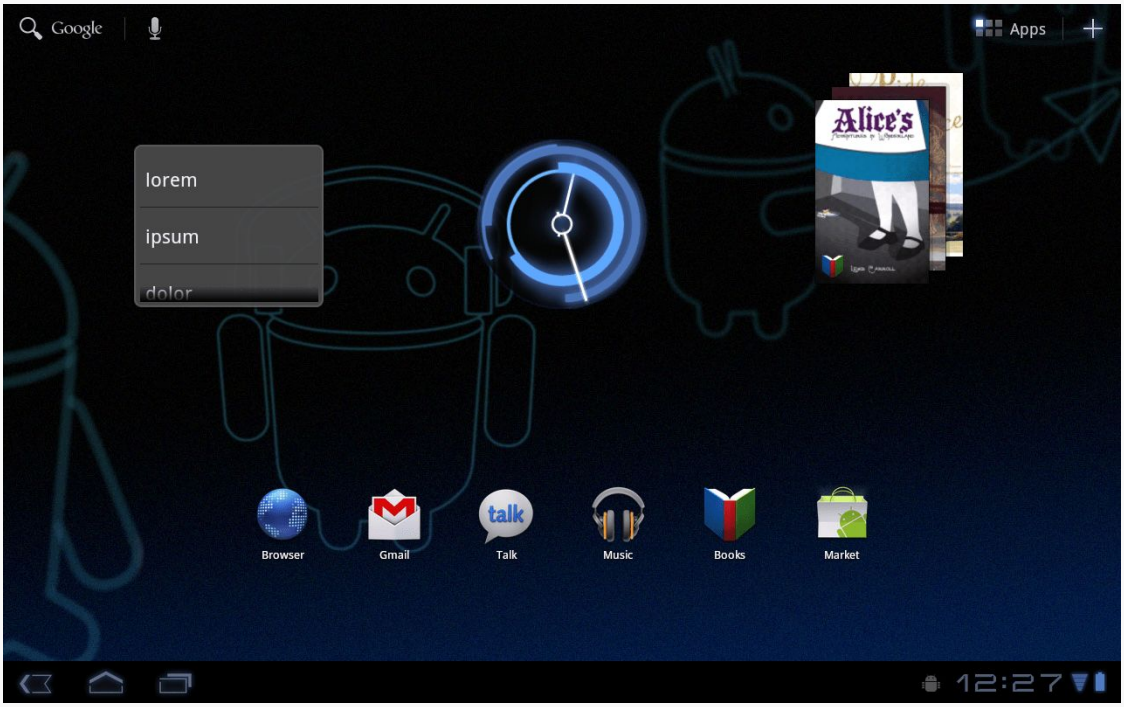


Figure 582: A XOOM home screen, showing the LoremWidget on the left

The `ListView` is live and can be scrolled. Tapping an entry brings up the corresponding `Toast`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 583: A XOOM home screen, showing the LoremWidget on the left

The above image illustrates that a `Toast` is not a great UI choice on a tablet, given the relative size of the `Toast` compared to the screen. Users will be far more likely to miss the `Toast` than ever before.

If the user long-taps on the app widget, they will be able to reposition it. On Android 3.1 and beyond, when they lift their finger after the long-tap, the app widget will show resize handles on the sides designated by your `android:resizeMode` attribute:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

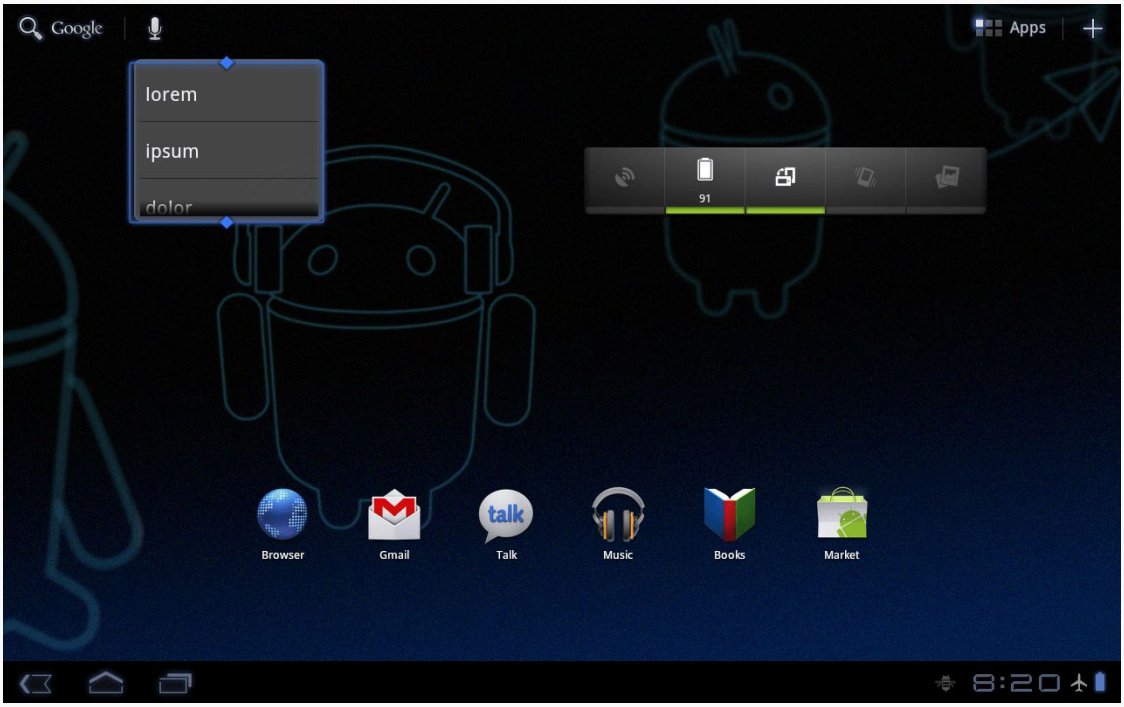


Figure 584: A phone home screen, showing the LoremWidget on the left, with resize handles

The user can then drag those handles to expand or shrink the app widget in the specified dimensions:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

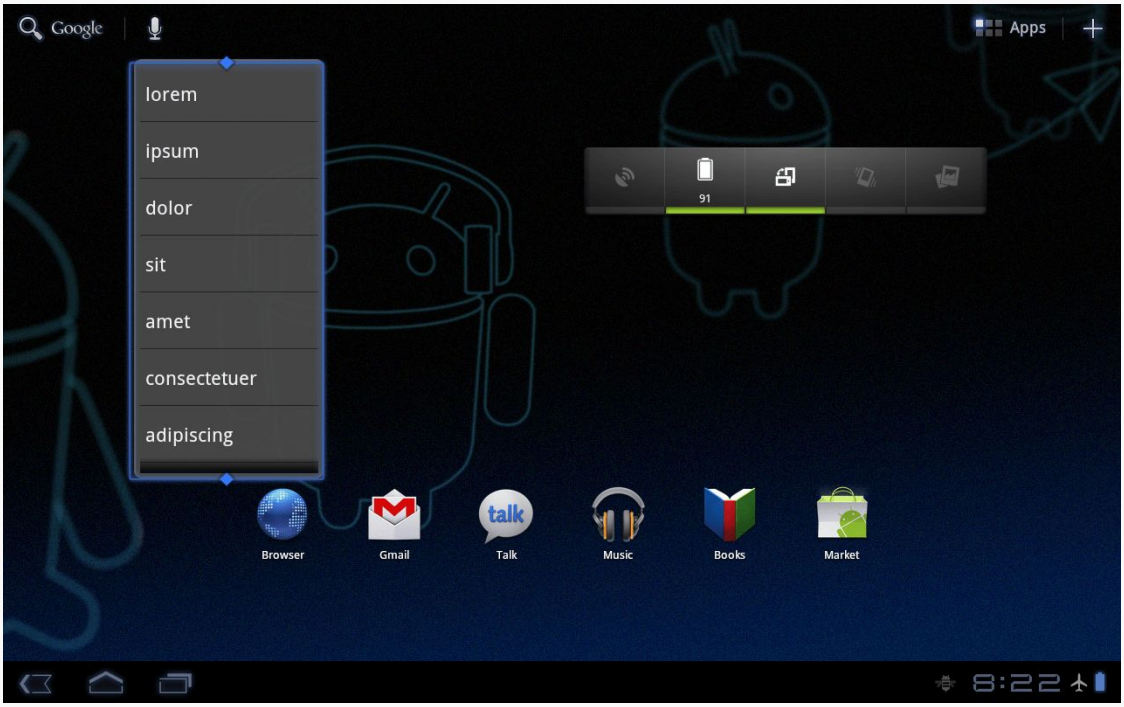


Figure 585: The resized LoremWidget

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Data Storage and Retrieval

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Content Provider Theory

Android publishes data to you via an abstraction known as a "content provider". Access to contacts and the call log, for example, are given to you via a set of content providers. In a few places, Android expects you to supply a content provider, such as for integrating your own search suggestions with the Android Quick Search Box. And, content providers are one way for you to supply data to third party applications, or to consume information from third party applications. As such, content providers have the potential to be something you would encounter frequently, even if in practice they do not seem used much.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on working with local databases.

Using a Content Provider

Any `Uri` in Android that begins with the `content://` scheme represents a resource served up by a content provider. Content providers offer data encapsulation using `Uri` instances as handles – you neither know nor care where the data represented by the `Uri` comes from, so long as it is available to you when needed. The data could be stored in a SQLite database, or in flat files, or retrieved off a device, or be stored on some far-off server accessed over the Internet.

Given a `Uri`, you may be able to perform basic CRUD (create, read, update, delete) operations using a content provider. `Uri` instances can represent either collections or individual pieces of content. Given a collection `Uri`, you may be able to create new pieces of content via insert operations. Given an instance `Uri`, you may be able to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

read data represented by the `Uri`, update that data, or delete the instance outright. Or, given a `Uri`, you may be able to open up a handle to what amounts to a file, that you can read and, possibly, write to.

These are all phrased as "may" because the content provider system is a facade. The actual implementation of a content provider dictates what you can and cannot do, and not all content providers will support all capabilities.

Pieces of Me

The simplified model of the construction of a content `Uri` is the scheme, the namespace of data, and, optionally, the instance identifier, all separated by slashes in URL-style notation. The scheme of a content Uri is always `content://`.

So, a content `Uri` of `content://constants/5` represents the constants instance with an identifier of `5`.

The combination of the scheme and the namespace is known as the "base Uri" of a content provider, or a set of data supported by a content provider. In the example above, `content://constants` is the base `Uri` for a content provider that serves up information about "constants" (in this case, physical constants).

The base `Uri` can be more complicated. For example, if the base `Uri` for contacts were `content://contacts/people`, the contacts content provider may serve up other data using other base `Uri` values.

The base `Uri` represents a collection of instances. The base `Uri` combined with an instance identifier (e.g., 5) represents a single instance.

Most of the Android APIs expect these to be `Uri` objects, though in common discussion, it is simpler to think of them as strings. The `Uri.parse()` static method creates a `Uri` out of the string representation.

Getting a Handle

So, where do these `Uri` instances come from?

The most popular starting point, if you know the type of data you want to work with, is to get the base `Uri` from the content provider itself in code. For example, `CONTENT_URI` is the base `Uri` for contacts represented as people — this maps to `content://contacts/people`. If you just need the collection, this `Uri` works as-is; if

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

you need an instance and know its identifier, you can call `addId()` on the `Uri` to inject it, so you have a `Uri` for the instance.

You might also get `Uri` instances handed to you from other sources, such as getting `Uri` handles for contacts via sub-activities responding to `ACTION_PICK` intents. In this case, the `Uri` is truly an opaque handle... unless you decide to pick it apart using the various getters on the `Uri` class.

You can also hard-wire literal `String` objects (e.g., `"content://contacts/people"`) and convert them into `Uri` instances via `Uri.parse()`. This is not an ideal solution, as the base `Uri` values could conceivably change over time. For example, the contacts content provider's base `Uri` is no longer `content://contacts/people` due to an overhaul of that subsystem. However, when you integrate with content providers from third parties, most likely you will not have a choice but to "hard-wire" in the content `Uri` based on a string.

The Database-Style API

Of the two flavors of API that a content provider may support, the database-style API is more prevalent. Using a `ContentResolver`, you can perform standard "CRUD" operations (create, read, update, delete) using what looks like a SQL interface.

Makin' Queries

Given a base `Uri`, you can run a query to return data out of the content provider related to that `Uri`. This has much of the feel of SQL: you specify the "columns" to return, the constraints to determine which "rows" to return, a sort order, etc. The difference is that this request is being made of a content provider, not directly of some database (e.g., SQLite).

You have two main options for running a query:

1. Use the `query()` method on `ContentResolver` from some sort of background thread
2. Use a `CursorLoader`, as is discussed in an upcoming chapter

The standard `query()` method on `ContentResolver` takes five parameters:

- The base `Uri` of the content provider to query, or the instance `Uri` of a specific object to query

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- An array of properties (think "columns") from that content provider that you want returned by the query
- A constraint statement, functioning like a SQL `WHERE` clause
- An optional set of parameters to bind into the constraint clause, replacing any `?` that appear there
- An optional sort statement, functioning like a SQL `ORDER BY` clause

This method returns a `Cursor` object, which you can use to retrieve the data returned by the query.

This will hopefully make more sense given an example. This chapter shows some sample bits of code from the `ContentProvider/ConstantsPlus` sample project. This is the same basic application as was first shown back in the chapter on database access, but rewritten to pull the database logic into a content provider, which is then used by the activity.

The activity starts off, in `onCreate()`, by executing a `LoadCursorTask`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  new LoadCursorTask().execute();
}
```

In `doInBackground()`, `LoadCursorTask` calls a `doQuery()` method on the activity, which in turn uses a `ContentResolver` to query our `ContentProvider`:

```
private Cursor doQuery() {
  return(getContentResolver().query(Provider.Constants.CONTENT_URI,
                                    PROJECTION, null, null, null));
}
```

In the call to `query()`, we provide:

1. The `Uri` passed into the activity by the caller (`CONTENT_URI`), in this case representing the collection of physical constants managed by the content provider
2. A list of properties to retrieve
3. Three `null` values, indicating that we do not need a constraint clause (the `Uri` represents the instance we need), nor parameters for the constraint, nor a sort order (we should only get one entry back)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The biggest "magic" here is the list of properties. The lineup of what properties are possible for a given content provider should be provided by the documentation (or source code) for the content provider itself. In this case, we define logical values on the `Provider` content provider implementation class that represent the various properties (namely, the unique identifier, the display name or title, and the value of the constant).

Adapting to the Circumstances

Now that we have a `Cursor` via `query()`, we have access to the query results and can do whatever we want with them. You might, for example, manually extract data from the `Cursor` to populate widgets or other objects.

However, if the goal of the query was to return a list from which the user should choose an item, you probably should consider using `SimpleCursorAdapter`. This class bridges between the `Cursor` and a selection widget, such as a `ListView` or `Spinner`. Pour the `Cursor` into a `SimpleCursorAdapter`, hand the adapter off to the widget, and you are set — your widget will show the available options.

After executing the `query()` and getting the `Cursor`, the `LoadCursorTask` in `ConstantsBrowser`, in its `onPostExecute()` method, creates a `SimpleCursorAdapter` with the following parameters:

1. The activity (or other `Context`) creating the adapter; in this case, the `ConstantsBrowser` itself
2. The identifier for a layout to be used for rendering the list entries (`R.layout.row`)
3. The cursor (`constantsCursor`)
4. The properties to pull out of the cursor and use for configuring the list entry View instances (`TITLE` and `VALUE`)
5. The corresponding identifiers of `TextView` widgets in the list entry layout that those properties should go into (`R.id.title` and `R.id.value`)

```
new SimpleCursorAdapter(this,
                        R.layout.row, constantsCursor,
                        new String[] {Provider.Constants.TITLE,
                                      Provider.Constants.VALUE},
                        new int[] {R.id.title, R.id.value});
```

If you need more control over the views than you can reasonably achieve with the stock view construction logic, subclass `SimpleCursorAdapter` and override

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`getView()` to create your own widgets to go into the list, as demonstrated earlier in this book.

The complexity regarding which `SimpleCursorAdapter` constructor to use is because Google deprecated the old constructor in API Level 11, replacing it with a new constructor... that just takes a flag as an additional parameter. Normally, that flag is `0`. But, since the new constructor does not exist on API Level 10 and below, we create the `SimpleCursorAdapter` using the proper constructor based on API level.

And, of course, you can manually manipulate the `Cursor` (e.g., `moveToFirst()`, `getString()`), just like you can with a database `Cursor`.

Give and Take

Of course, content providers would be astonishingly weak if you couldn't add or remove data from them, and were instead limited to only update what is there. Fortunately, content providers offer these abilities as well.

To insert data into a content provider, you have two options available on the `ContentProvider` interface (available through `getContentResolver()` to your activity):

- Use `insert()` with a collection `Uri` and a `ContentValues` structure describing the initial set of data to put in the row
- Use `bulkInsert()` with a collection `Uri` and an array of `ContentValues` structures to populate several rows at once

The `insert()` method returns a `Uri` for you to use for future operations on that new object. The `bulkInsert()` method returns the number of created rows; you would need to do a query to get back at the data you just inserted.

For example, if the user chooses our "Add" overflow item, we pop up a dialog to collect a new constant:

```
private void add() {
  View addView=getLayoutInflater().inflate(R.layout.add_edit, null);
  AlertDialog.Builder builder=new AlertDialog.Builder(this);

  builder.setTitle(R.string.add_title).setView(addView)
         .setPositiveButton(R.string.ok, this)
         .setNegativeButton(R.string.cancel, null).show();
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Then, if the user taps the "OK" button in the dialog, our `onClick()` listener is called, where we collect the entered values from the user, pour them into a `ContentValues` structure, and pass that to an `InsertTask`:

```
@Override
public void onClick(DialogInterface dialog, int which) {
  ContentValues values=new ContentValues(2);
  AlertDialog dlg=(AlertDialog)dialog;
  EditText title=(EditText)dlg.findViewById(R.id.title);
  EditText value=(EditText)dlg.findViewById(R.id.value);

  values.put(DatabaseHelper.TITLE, title.getText().toString());
  values.put(DatabaseHelper.VALUE, value.getText().toString());

  new InsertTask().execute(values);
}
```

`InsertTask`, in its `doInBackground()` method, calls `insert()` on a `ContentResolver` to insert this row:

```
  @Override
  protected Void doInBackground(ContentValues... values) {
    getContentResolver().insert(Provider.Constants.CONTENT_URI, values[0]);

    constantsCursor=doQuery();
    constantsCursor.getCount();

    return(null);
  }
```

Notice that we also call `doQuery()` again. That is because our `Cursor` is now out of date, and we need to obtain a fresh `Cursor` with fresh results. We also call `getCount()` on the `Cursor` — both in `InsertTask` and in `LoadCursorTask` — just to be sure that the `Cursor` has loaded all of its data, while we are safely on the background thread.

The `onPostExecute()` implementation in `InsertTask` uses `changeCursor()` on `CursorAdapter()` to swap in our new `Cursor` for any old one:

```
  @Override
  public void onPostExecute(Void arg0) {
    ((CursorAdapter)getListAdapter()).changeCursor(constantsCursor);
  }
```

To delete one or more rows from the content provider, use the `delete()` method on `ContentResolver`. This works akin to a SQL `DELETE` statement and takes three parameters:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- A `Uri` representing the collection (or instance) from which you wish to delete rows
- A constraint statement, functioning like a SQL `WHERE` clause, to determine which rows should be deleted
- An optional set of parameters to bind into the constraint clause, replacing any `?` that appear there

The File System-Style API

Sometimes, what you are trying to retrieve does not look like a set of rows and columns, but rather looks like a file. For example, the `MediaStore` content provider manages the index of all music, video, and image files available on external storage, and you can use `MediaStore` to open up any such file you find.

Some content providers, like `MediaStore`, support both the database-style and file system-style APIs — you query to find media that matches your criteria, then can open some file that matches. Other content providers might only support the file system-style API.

Given a `Uri` that represents some file managed by the content provider, you can use `openInputStream()` and `openOutputStream()` on a `ContentResolver` to access an `InputStream` or `OutputStream`, respectively. Note, though, that not all content providers may support both modes. For example, a content provider that serves files stored inside the application (e.g., assets in the APK file), you will not be able to get an `OutputStream` to modify the content.

Building Content Providers

Building a content provider is probably a very tedious task. There are many requirements of a content provider, in terms of methods to implement and public data members to supply. And, until you try using it, you have no great way of telling if you did any of it correctly (versus, say, building an activity and getting validation errors from the resource compiler).

That being said, building a content provider is of huge importance if your application wishes to make data available to other applications. If your application is keeping its data solely to itself, you may be able to avoid creating a content provider, just accessing the data directly from your activities. But, if you want your data to possibly be used by others — for example, you are building a feed reader and you

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

want other programs to be able to access the feeds you are downloading and caching — then a content provider is right for you.

First, Some Dissection

The content `Uri` is the linchpin behind accessing data inside a content provider. When using a content provider, all you really need to know is the provider's base `Uri`; from there you can run queries as needed, or construct a `Uri` to a specific instance if you know the instance identifier.

When building a content provider, though, you need to know a bit more about the innards of the content `Uri`.

A content `Uri` has two to four pieces, depending on situation:

1. It always has a scheme (`content://`), indicating it is a content `Uri` instead of a `Uri` to a Web resource (`http://`).
2. It always has an authority, which is the first path segment after the scheme. The authority is a unique string identifying the content provider that handles the content associated with this `Uri`.
3. It may have a data type path, which is the list of path segments after the authority and before the instance identifier (if any). The data type path can be empty, if the content provider only handles one type of content. It can be a single path segment (`foo`) or a chain of path segments (`foo/bar/goo`) as needed to handle whatever data access scenarios the content provider requires.
4. It may have an instance identifier, which is an integer identifying a specific piece of content. A content `Uri` without an instance identifier refers to the collection of content represented by the authority (and, where provided, the data path).

For example, a content `Uri` could be as simple as `content://sekrits`, which would refer to the collection of content held by whatever content provider was tied to the `sekrits` authority (e.g., `SecretsProvider`). Or, it could be as complex as `content://sekrits/card/pin/17`, which would refer to a piece of content (identified as `17`) managed by the `sekrits` content provider that is of the data type `card/pin`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Next, Some Typing

Next, you need to come up with some MIME types corresponding with the content your content provider will provide.

Android uses both the content `Uri` and the MIME type as ways to identify content on the device. A collection content `Uri` — or, more accurately, the combination of authority and data type path – should map to a pair of MIME types. One MIME type will represent the collection; the other will represent an instance. These map to the `Uri` patterns above for no-identifier and identifier, respectively. As you saw earlier in this book, you can fill in a MIME type into an `Intent` to route the `Intent` to the proper activity (e.g., `ACTION_PICK` on a collection MIME type to call up a selection activity to pick an instance out of that collection).

The collection MIME type should be of the form `vnd.X.cursor.dir/Y`, where `X` is the name of your firm, organization, or project, and `Y` is a dot-delimited type name. So, for example, you might use `vnd.tlagency.cursor.dir/sekrits.card.pin` as the MIME type for your collection of secrets.

The instance MIME type should be of the form `vnd.X.cursor.item/Y`, usually for the same values of `X` and `Y` as you used for the collection MIME type (though that is not strictly required).

Implementing the Database-Style API

Just as an activity and a receiver are both Java classes, so is a content provider. So, the big step in creating a content provider is crafting its Java class, with a base class of `ContentProvider`.

In your subclass of `ContentProvider`, you are responsible for implementing five methods that, when combined, perform the services that a content provider is supposed to offer to activities wishing to create, read, update, or delete content via the database-style API.

Implement onCreate()

As with an activity, the main entry point to a content provider is `onCreate()`. Here, you can do whatever initialization you want. In particular, here is where you should lazy-initialize your data store. For example, if you plan on storing your data in such-and-so directory on an SD card, with an XML file serving as a “table of contents”, you

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

should check and see if that directory and XML file are there and, if not, create them so the rest of your content provider knows they are out there and available for use.

Similarly, if you have rewritten your content provider sufficiently to cause the data store to shift structure, you should check to see what structure you have now and adjust it if what you have is out of date.

Implement query()

As one might expect, the `query()` method is where your content provider gets details on a query some activity wants to perform. It is up to you to actually process said query.

The query method gets, as parameters:

1. A `Uri` representing the collection or instance being queried
2. A `String` array representing the list of properties that should be returned
3. A `String` representing what amounts to a SQL `WHERE` clause, constraining which instances should be considered for the query results
4. A `String` array representing values to "pour into" the `WHERE` clause, replacing any `?` found there
5. A `String` representing what amounts to a SQL `ORDER BY` clause

You are responsible for interpreting these parameters however they make sense and returning a `Cursor` that can be used to iterate over and access the data.

As you can imagine, these parameters are aimed towards people using a SQLite database for storage. You are welcome to ignore some of these parameters (e.g., you elect not to try to roll your own SQL `WHERE` clause parser), but you need to document that fact so activities only attempt to query you by instance `Uri` and not by using parameters that you elect to ignore.

Implement insert()

Your `insert()` method will receive a `Uri` representing the collection and a `ContentValues` structure with the initial data for the new instance. You are responsible for creating the new instance, filling in the supplied data, and returning a `Uri` to the new instance.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Implement update()

Your `update()` method gets the `Uri` of the instance or collection to change, a `ContentValues` structure with the new values to apply, a String for a SQL `WHERE` clause, and a `String` array with parameters to use to replace `?` found in the `WHERE` clause. Your responsibility is to identify the instance(s) to be modified (based on the `Uri` and `WHERE` clause), then replace those instances' current property values with the ones supplied.

This will be annoying, unless you are using SQLite for storage. Then, you can pretty much pass all the parameters you received to the `update()` call to the database, though the `update()` call will vary slightly depending on whether you are updating one instance or several.

Implement delete()

As with `update()`, `delete()` receives a `Uri` representing the instance or collection to work with and a `WHERE` clause and parameters. If the activity is deleting a single instance, the `Uri` should represent that instance and the `WHERE` clause may be null. But, the activity might be requesting to delete an open-ended set of instances, using the `WHERE` clause to constrain which ones to delete.

As with `update()`, though, this is simple if you are using SQLite for database storage (sense a theme?). You can let it handle the idiosyncrasies of parsing and applying the `WHERE` clause — all you have to do is call `delete()` on the database.

Implement getType()

The last method you need to implement is `getType()`. This takes a `Uri` and returns the MIME type associated with that `Uri`. The `Uri` could be a collection or an instance `Uri`; you need to determine which was provided and return the corresponding MIME type.

Update the Manifest

The glue tying the content provider implementation to the rest of your application resides in your `AndroidManifest.xml` file. Simply add a `<provider>` element as a child of the `<application>` element, such as:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.constants"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <provider
      android:name=".Provider"
      android:authorities="com.commonsware.android.constants.Provider"
      android:exported="false"/>

    <activity
      android:name=".ConstantsBrowser"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

The `android:name` property is the name of the content provider class, with a leading dot to indicate it is in the stock namespace for this application's classes (just like you use with activities).

The `android:authorities` property should be a semicolon-delimited list of the authority values supported by the content provider. Recall, from earlier in this chapter, that each content `Uri` is made up of a scheme, authority, data type path, and instance identifier. Each authority from each `CONTENT_URI` value should be included in the `android:authorities` list.

Now, when Android encounters a content `Uri`, it can sift through the providers registered through manifests to find a matching authority. That tells Android which

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

application and class implements the content provider, and from there Android can bridge between the calling activity and the content provider being called.

Add Notify-On-Change Support

A feature that your content provider can offer to its clients is notify-on-change support. This means that your content provider will let clients know if the data for a given content `Uri` changes.

For example, suppose you have created a content provider that retrieves RSS and Atom feeds from the Internet based on the user's feed subscriptions (via OPML, perhaps). The content provider offers read-only access to the contents of the feeds, with an eye towards several applications on the phone using those feeds versus everyone implementing their own feed poll-fetch-and-cache system. You have also implemented a service that will get updates to those feeds asynchronously, updating the underlying data store. Your content provider could alert applications using the feeds that such-and-so feed was updated, so applications using that specific feed can refresh and get the latest data.

On the content provider side, to do this, call `notifyChange()` on your `ContentResolver` instance (available in your content provider via `getContext().getContentResolver()`). This takes two parameters: the `Uri` of the piece of content that changed and the `ContentObserver` that initiated the change. In many cases, the latter will be `null`; a non-`null` value simply means that the observer that initiated the change will not be notified of its own changes.

On the content consumer side, an activity can call `registerContentObserver()` on its `ContentResolver` (via `getContentResolver()`). This ties a `ContentObserver` instance to a supplied `Uri` — the observer will be notified whenever `notifyChange()` is called for that specific `Uri`. When the consumer is done with the `Uri`, `unregisterContentObserver()` releases the connection.

Implementing the File System-Style API

If you want consumers of your `ContentProvider` to be able to call `openInputStream()` or `openOutputStream()` on a `Uri`, you will need to implement the `openFile()` method. This method is optional — if you are not supporting `openInputStream()` or `openOutputStream()`, you do not need to implement `openFile()` at all.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `openFile()` method returns a curious object called a `ParcelFileDescriptor`. Given that, the `ContentResolver` can obtain the `InputStream` or `OutputStream` that was requested. There are various static methods on `ParcelFileDescriptor` to create instances of it, such as an `open()` method that takes a `File` object as the first parameter. Note that this works for both files on external storage and files within your own project's app-local file storage (e.g., `getFilesDir()`).

Note that you are welcome to also implement `onCreate()`, if you wish to do some initialization when the content provider starts up. Also, you will have to provide do-nothing implementations of `query()`, `insert()`, `update()`, and `delete()`, as those methods are mandatory in `ContentProvider` subclasses, even if you do not plan to support them.

Issues with Content Providers

Content providers are not without their issues.

The biggest complaint seems to be the lack of an `onDestroy()` companion to the `onCreate()` method you can implement. Hence, if you open a database in `onCreate()`, you close it... never. Sometimes, you can alleviate this by initializing things on demand and releasing them immediately, such as opening a database as part of `insert()` and closing it within the same method. This does not always work, however — for example, you cannot close the database you query in `query()`, since the `Cursor` you return would become invalid.

The fact that `ContentProvider` is effectively a facade means that a consumer of a `ContentProvider` has no idea what to expect. It is up to documentation to explain what `Uri` values can be used, what columns can be returned, what query syntax is supported, and so on. And, the fact that it is a facade means that much of the richness of the SQLite interface is lost, such as `GROUP BY`. To top it off, the API supported by `ContentProvider` is rather limited — if what you want to share does not look like a database and does not look like a file, it may be difficult to force it into the `ContentProvider` API.

However, perhaps the biggest problem is that, by default, content providers are exported, meaning they can be accessed by other processes (third party applications or the Android OS). Sometimes this is desired. Sometimes, it is not. You need to set `android:exported` to be `false` on your manifest entry for the content provider if you want to keep the provider private to your application. This is the inverse of all other components, which are private by default, unless they have an `<intent-filter>`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that API Level 17 changes the default — if your `android:targetSdkVersion` is set to 17 or higher, `android:exported` is `false` by default, not `true` as before.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Content Provider Implementation Patterns

The previous chapter focused on the concepts, classes, and methods behind content providers. This chapter more closely examines some implementations of content providers, organized into simple patterns.

Prerequisites

Understanding this chapter requires that you have read the preceding chapter, along with the chapter on permissions.

The Single-Table Database-Backed Content Provider

The simplest database-backed content provider is one that only attempts to expose a single table's worth of data to consumers. The `CallLog` content provider works this way, for example.

Step #1: Create a Provider Class

We start off with a custom subclass of `ContentProvider`, named, cunningly enough, Provider. Here we need the database-style API methods: `query()`, `insert()`, `update()`, `delete()`, and `getType()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

onCreate()

Here is the `onCreate()` method for `Provider`, from the [ContentProvider/ConstantsPlus] sample application:

```
  @Override
  public boolean onCreate() {
    db=new DatabaseHelper(getContext());

    return((db == null) ? false : true);
  }
```

While that does not seem all that special, the "magic" is in the private `DatabaseHelper` object, a fairly conventional `SQLiteOpenHelper` implementation:

```
package com.commonsware.android.constants;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.hardware.SensorManager;

class DatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="constants.db";
  static final String TITLE="title";
  static final String VALUE="value";

  public DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, 1);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    Cursor c=db.rawQuery("SELECT name FROM sqlite_master WHERE type='table' AND
name='constants'", null);

    try {
      if (c.getCount()==0) {
        db.execSQL("CREATE TABLE constants (_id INTEGER PRIMARY KEY
AUTOINCREMENT, title TEXT, value REAL);");

        ContentValues cv=new ContentValues();

        cv.put(Provider.Constants.TITLE, "Gravity, Death Star I");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_DEATH_STAR_I);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Earth");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_EARTH);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Jupiter");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_JUPITER);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Mars");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_MARS);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Mercury");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_MERCURY);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Moon");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_MOON);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Neptune");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_NEPTUNE);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Pluto");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_PLUTO);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Saturn");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_SATURN);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Sun");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_SUN);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, The Island");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_THE_ISLAND);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Uranus");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_URANUS);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Venus");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_VENUS);
        db.insert("constants", Provider.Constants.TITLE, cv);
      }
    }
    finally {
      c.close();
    }
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android.util.Log.w("Constants", "Upgrading database, which will destroy all
old data");
    db.execSQL("DROP TABLE IF EXISTS constants");
    onCreate(db);
  }
}
```

Note that we are creating the `DatabaseHelper` in `onCreate()` and are never closing it. That is because there is no `onDestroy()` (or equivalent) method in a `ContentProvider`. While we might be tempted to open and close the database on every operation, that will not work, as we cannot close the database and still hand back a live `Cursor` from the database. Hence, we leave it open and assume that SQLite's transactional nature will ensure that our database is not corrupted when Android shuts down the `ContentProvider`.

query()

For SQLite-backed storage providers like this one, the `query()` method implementation should be largely boilerplate. Use a `SQLiteQueryBuilder` to convert the various parameters into a single SQL statement, then use `query()` on the builder to actually invoke the query and give you a `Cursor` back. The `Cursor` is what your `query()` method then returns.

For example, here is `query()` from `Provider`:

```
  @Override
  public Cursor query(Uri url, String[] projection, String selection,
                      String[] selectionArgs, String sort) {
    SQLiteQueryBuilder qb=new SQLiteQueryBuilder();

    qb.setTables(TABLE);

    String orderBy;

    if (TextUtils.isEmpty(sort)) {
      orderBy=Constants.DEFAULT_SORT_ORDER;
    }
    else {
      orderBy=sort;
    }

    Cursor c=
        qb.query(db.getReadableDatabase(), projection, selection,
                 selectionArgs, null, null, orderBy);

    c.setNotificationUri(getContext().getContentResolver(), url);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(c);
  }
```

We create a `SQLiteQueryBuilder` and pour the query details into the builder, notably the name of the table that we query against and the sort order (substituting in a default sort if the caller did not request one). When done, we use the `query()` method on the builder to get a `Cursor` for the results. We also tell the resulting `Cursor` what `Uri` was used to create it, for use with the content observer system.

The `query()` implementation, like many of the other methods on `Provider`, delegates much of the `Provider`-specific information to private methods, such as:

1. the name of the table (`getTableName()`)
2. the default sort order (`getDefaultSortOrder()`)

insert()

Since this is a SQLite-backed content provider, once again, the implementation is mostly boilerplate: validate that all required values were supplied by the activity, merge your own notion of default values with the supplied data, and call `insert()` on the database to actually create the instance.

For example, here is `insert()` from `Provider`:

```
  @Override
  public Uri insert(Uri url, ContentValues initialValues) {
    long rowID=
        db.getWritableDatabase().insert(TABLE, Constants.TITLE,
                                        initialValues);

    if (rowID > 0) {
      Uri uri=
          ContentUris.withAppendedId(Provider.Constants.CONTENT_URI,
                                     rowID);
      getContext().getContentResolver().notifyChange(uri, null);

      return(uri);
    }

    throw new SQLException("Failed to insert row into " + url);
  }
```

The pattern is the same as before: use the provider particulars plus the data to be inserted to actually do the insertion.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

update()

Here is `update()` from `Provider`:

```
  @Override
  public int update(Uri url, ContentValues values, String where,
                    String[] whereArgs) {
    int count=
        db.getWritableDatabase()
          .update(TABLE, values, where, whereArgs);

    getContext().getContentResolver().notifyChange(url, null);

    return(count);
  }
```

In this case, updates are always applied across the entire collection, though we could have a smarter implementation that supported updating a single instance via an instance `Uri`.

delete()

Similarly, here is `delete()` from `Provider`:

```
  @Override
  public int delete(Uri url, String where, String[] whereArgs) {
    int count=db.getWritableDatabase().delete(TABLE, where, whereArgs);

    getContext().getContentResolver().notifyChange(url, null);

    return(count);
  }
```

This is almost a clone of the `update()` implementation described above.

getType()

The last method you need to implement is `getType()`. This takes a `Uri` and returns the MIME type associated with that `Uri`. The `Uri` could be a collection or an instance `Uri`; you need to determine which was provided and return the corresponding MIME type.

For example, here is `getType()` from `Provider`:

```
  @Override
  public String getType(Uri url) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (isCollectionUri(url)) {
      return("vnd.commonsware.cursor.dir/constant");
    }

    return("vnd.commonsware.cursor.item/constant");
  }
```

Step #2: Supply a Uri

You may wish to add a public static member... somewhere, containing the `Uri` for each collection your content provider supports, for use by your own application code. Typically, this is a public static final `Uri` put on the content provider class itself:

```
    public static final Uri CONTENT_URI=
        Uri.parse("content://com.commonsware.android.constants.Provider/
constants");
```

You may wish to use the same namespace for the content `Uri` that you use for your Java classes, to reduce the chance of collision with others.

Bear in mind that if you intend for third parties to access your content provider, they will not have access to this public static data member, as your class is not in their project. Hence, you will need to publish the string representation of this `Uri` that they can hard-wire into their application.

Step #3: Declare the "Columns"

Remember those "columns" you referenced when you were using a content provider, in the previous chapter? Well, you may wish to publish public static values for those too for your own content provider.

Specifically, you may want a public static class implementing `BaseColumns` that contains your available column names, such as this example from `Provider`:

```
  public static final class Constants implements BaseColumns {
    public static final Uri CONTENT_URI=
        Uri.parse("content://com.commonsware.android.constants.Provider/
constants");
    public static final String DEFAULT_SORT_ORDER="title";
    public static final String TITLE="title";
    public static final String VALUE="value";
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Since we are using SQLite as a data store, the values for the column name constants should be the corresponding column names in the table, so you can just pass the projection (array of columns) to SQLite on a `query()`, or pass the `ContentValues` on an `insert()` or `update()`.

Note that nothing in here stipulates the types of the properties. They could be strings, integers, or whatever. The biggest limitation is what a `Cursor` can provide access to via its property getters. The fact that there is nothing in code that enforces type safety means you should document the property types well, so people attempting to use your content provider know what they can expect.

Step #4: Update the Manifest

Finally, we need to add the provider to the `AndroidManifest.xml` file, by adding a `<provider>` element as a child of the `<application>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.constants"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <provider
      android:name=".Provider"
      android:authorities="com.commonsware.android.constants.Provider"
      android:exported="false"/>

    <activity
      android:name=".ConstantsBrowser"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  </application>

</manifest>
```

The Local-File Content Provider

Implementing a content provider that supports serving up files based on `Uri` values is similar, and generally simpler, than creating a content provider for the database-style API. In this section, we will examine the `ContentProvider/Files` sample project. This project demonstrates a common use of the filesystem-style API: serving files from internal storage to third-party applications (who, by default, cannot read your internally-stored files).

Note that this sample project will only work on devices that have an application capable of viewing PDF files accessed via `content://` `Uri` values.

The FileProvider Class

Our `ContentProvider` is named `FileProvider`. However, most of the logic is contained in an `AbstractFileProvider` that will be used for a handful of sample apps in this chapter. We will look at both of those classes, focusing first on the `FileProvider`.

onCreate()

We have an `onCreate()` method. In many cases, this would not be needed for this sort of provider. After all, there is no database to open. In this case, we use `onCreate()` to copy the file(s) out of assets into the app-local file store. In principle, this would allow our application code to modify these files as the user uses the app (versus the unmodifiable editions in `assets/`).

```
  @Override
  public boolean onCreate() {
    File f=new File(getContext().getFilesDir(), "test.pdf");

    if (!f.exists()) {
      AssetManager assets=getContext().getResources().getAssets();

      try {
        copy(assets.open("test.pdf"), f);
      }
      catch (IOException e) {
        Log.e("FileProvider", "Exception copying from assets", e);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        return(false);
      }
    }

    return(true);
  }
```

This uses a static `copy()` method, inherited from `AbstractFileProvider`, that can copy an `InputStream` from an asset to a local `File`. We will take a peek at this later in this chapter.

openFile()

We need to implement `openFile()`, to return a `ParcelFileDescriptor` corresponding to the supplied `Uri`:

```
  @Override
  public ParcelFileDescriptor openFile(Uri uri, String mode)
                                                       throws
FileNotFoundException {
    File f=new File(getContext().getFilesDir(), uri.getPath());

    if (f.exists()) {
      return(ParcelFileDescriptor.open(f, parseMode(mode)));
    }

    throw new FileNotFoundException(uri.getPath());
  }
```

We are passed in a *nix-style string `mode`, which will be a value like `r` for read access, `wt` for write access (and truncate the file), etc. In API Level 19+, `ParcelFileDescriptor` has a convenience method for converting such modes into the equivalent `ParcelFileDescriptor` flag values. For older devices, you can simply use the `parseMode()` code that Google added:

```
  // following is from ParcelFileDescriptor source code
  // Copyright (C) 2006 The Android Open Source Project
  // (even though this method was added much after 2006...)

  private static int parseMode(String mode) {
    final int modeBits;
    if ("r".equals(mode)) {
      modeBits=ParcelFileDescriptor.MODE_READ_ONLY;
    }
    else if ("w".equals(mode) || "wt".equals(mode)) {
      modeBits=
          ParcelFileDescriptor.MODE_WRITE_ONLY
              | ParcelFileDescriptor.MODE_CREATE
              | ParcelFileDescriptor.MODE_TRUNCATE;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
    else if ("wa".equals(mode)) {
      modeBits=
          ParcelFileDescriptor.MODE_WRITE_ONLY
              | ParcelFileDescriptor.MODE_CREATE
              | ParcelFileDescriptor.MODE_APPEND;
    }
    else if ("rw".equals(mode)) {
      modeBits=
          ParcelFileDescriptor.MODE_READ_WRITE
              | ParcelFileDescriptor.MODE_CREATE;
    }
    else if ("rwt".equals(mode)) {
      modeBits=
          ParcelFileDescriptor.MODE_READ_WRITE
              | ParcelFileDescriptor.MODE_CREATE
              | ParcelFileDescriptor.MODE_TRUNCATE;
    }
    else {
      throw new IllegalArgumentException("Bad mode '" + mode + "'");
    }
    return modeBits;
  }
```

Our `openFile()` method then uses `parseMode()` in the call to the static `open()` method on `ParcelFileDescriptor`, which opens the file (with the desired access mode) and gives us our `ParcelFileDescriptor` back that we can return. If the file is not found, we can throw a `FileNotFoundException` to indicate that.

getDataLength()

`AbstractFileProvider` gives us a callback — `getDataLength()` — where we can indicate how big a file is, given its `Uri`. That information will be made available to clients consuming this stream. The default will be to indicate that the file size is unknown... and that usually works. However, if it is easy for you to determine the file size, do so, and it will increase the compatibility of your app with possible consumers.

In this case, determining the size of a local file is easy:

```
  protected long getDataLength(Uri uri) {
    File f=new File(getContext().getFilesDir(), uri.getPath());

    return(f.length());
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The AbstractFileProvider Class

`AbstractFileProvider` is designed to handle a lot of common boilerplate for streaming providers like the one provided in this sample.

getType()

Just as our database-style `ContentProvider` needed to implement `getType()` to provide a MIME type given a `Uri`, so too do our streaming providers. The difference is that a streaming provider usually wants to use "real" MIME types, values that third-party apps are likely to recognize. For example, a PDF file should use a MIME type of `application/pdf`, as that is what PDF viewing apps will expect.

Android has some convenience code for determining a likely MIME type. You can use `MimeTypeMap` to convert a file extension to a MIME type, or you can use `guessContentTypeFromName()` on`URLConnection` to get a MIME type for a URL. Both use the same underlying database — the difference is mostly a matter of whether you have a bare file extension already or not. So, the default implementation of `getType()` in `AbstractFileProvider` uses `guessContentTypeFromName()`:

```
@Override
public String getType(Uri uri) {
  return(URLConnection.guessContentTypeFromName(uri.toString()));
}
```

If you know that *your* MIME type is unlikely to be recognized by Android (e.g., you invented your own), a subclass of `AbstractFileProvider` could handle those cases, chaining to the superclass for other `Uri` values.

insert(), update(), and delete()

`ContentProvider` itself is `abstract`, requiring us to implement a variety of methods to satisfy the compiler. Three of them — `insert()`, `update()`, and `delete()` — have no role in a pure-streaming `ContentProvider`, so `AbstractFileProvider` has stub implementations:

```
@Override
public Uri insert(Uri uri, ContentValues initialValues) {
  throw new RuntimeException("Operation not supported");
}

@Override
public int update(Uri uri, ContentValues values, String where,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                    String[] whereArgs) {
  throw new RuntimeException("Operation not supported");
}

@Override
public int delete(Uri uri, String where, String[] whereArgs) {
  throw new RuntimeException("Operation not supported");
}
```

A `ContentProvider` that supports *both* the database-style and streaming APIs will need real implementations of those methods for the database operations, perhaps throwing an `Exception` for requests to insert, update, or delete a `Uri` that represents a stream.

query() and getFileName()

We also need to implement `query()`. You can get by with having this be a stub similar to `insert()` and kin. However, for better compatibility, you should have a more robust `query()` implementation, as it will be used by `ContentResolver` to retrieve two pieces of metadata about a `Uri`:

- What is a valid filename to use to represent this `Uri`, should we need a human-readable name? After all, a `ContentProvider` `Uri` does not have to represent a human-readable path, and so the last segment of that `Uri` could be a cryptic string of hex digits or something, not a filename.
- What is the length of the data that should be delivered by the stream?

`query()` will be called with a projection that contains either `OpenableColumns.DISPLAY_NAME`, `OpenableColumns.SIZE`, or both. A streaming `ContentProvider` ideally supports returning a `Cursor` with this data. The `AbstractFileProvider` implementation of `query()` handles this for us:

```
private final static String[] OPENABLE_PROJECTION= {
    OpenableColumns.DISPLAY_NAME, OpenableColumns.SIZE };

@Override
public Cursor query(Uri uri, String[] projection, String selection,
                    String[] selectionArgs, String sortOrder) {
  if (projection == null) {
    projection=OPENABLE_PROJECTION;
  }

  final MatrixCursor cursor=new MatrixCursor(projection, 1);

  MatrixCursor.RowBuilder b=cursor.newRow();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    for (String col : projection) {
      if (OpenableColumns.DISPLAY_NAME.equals(col)) {
        b.add(getFileName(uri));
      }
      else if (OpenableColumns.SIZE.equals(col)) {
        b.add(getDataLength(uri));
      }
      else { // unknown, so just add null
        b.add(null);
      }
    }

    return(cursor);
  }
```

If the supplied `projection` is `null`, we assume that the caller wants the standard `OpenableColumns`; otherwise, we will use the supplied `projection`.

Our return value will be a `MatrixCursor`. This amounts to a `Cursor` interface on a two-dimensional array, where you build up the rows in that array via a `MatrixCursor.RowBuilder`. In our case, there will only be one such row, for the relevant values for the file to be streamed in support of the requested `Uri`.

We iterate over the columns in the projection, calling out to `getFileName()` and `getDataLength()` methods for `OpenableColumns.DISPLAY_NAME` and `OpenableColumns.SIZE` respectively (and using `null` as the result for anything else). The default implementations of those methods return the last path segment of the `Uri` and `AssetFileDescriptor.UNKNOWN_LENGTH`, respectively:

```
  protected String getFileName(Uri uri) {
    return(uri.getLastPathSegment());
  }

  protected long getDataLength(Uri uri) {
    return(AssetFileDescriptor.UNKNOWN_LENGTH);
  }
```

Subclasses can override those as needed, as we saw with `getDataLength()` in the concrete `FileProvider` class.

copy()

`AbstractFileProvider` also has a convenience `copy()` static method that copies an `InputStream` to a `File`, used from the `FileProvider` `onCreate()` method:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  static protected void copy(InputStream in, File dst)
                                                    throws IOException {
    FileOutputStream out=new FileOutputStream(dst);
    byte[] buf=new byte[1024];
    int len;

    while ((len=in.read(buf)) >= 0) {
      out.write(buf, 0, len);
    }

    in.close();
    out.close();
  }
}
```

The Manifest

Finally, we need to add the provider to the `AndroidManifest.xml` file, by adding a `<provider>` element as a child of the `<application>` element, as with any other content provider:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.cp.files"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="FilesCPDemo"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>

    <provider
      android:name=".FileProvider"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      android:authorities="com.commonsware.android.cp.files"
      android:exported="true"/>
  </application>

</manifest>
```

Note, however, that we have `android:exported="true"` set in our `<provider>` element. This means that this content provider can be accessed from third-party apps or other external processes (e.g., the media framework for playing back videos).

Using this Provider

The activity is fairly trivial, simply creating an `ACTION_VIEW` `Intent` on our PDF file and starting up an activity for it, then finishing itself:

```
package com.commonsware.android.cp.files;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;

public class FilesCPDemo extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    startActivity(new Intent(Intent.ACTION_VIEW,
                             Uri.parse(FileProvider.CONTENT_URI
                                 + "test.pdf")));
    finish();
  }
}
```

Here, we use a `CONTENT_URI` published by `FileProvider` as the basis for identifying the file:

```
  public static final Uri CONTENT_URI=
      Uri.parse("content://com.commonsware.android.cp.files/");
```

The Protected Provider

The problem with the preceding example is that any app on the device, if it knows the right `Uri` to ask for, will be able to access the file. This may be desired, but often times it will not be. Instead, you may want to specifically indicate which apps, at specific points in time, can view the file.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Particularly if your objective is to start a third-party app to work with that file, setting up this sort of security is not that difficult. To see how that works, we will walk through the ContentProvider/GrantUriPermissions sample project. This is a clone of the ContentProvider/Files project with this extra security added on.

The way the defense works is by using Android's permission system. We will mark the ContentProvider as being not exported, then selectively grant that access to a specific Uri to the app that we want to view our file.

Step #1: Mark the Provider as Not Exported

Putting android:exported="false" on the <provider> element indicates that no app has the ability to make requests of your ContentProvider, except for specific cases where you authorize it:

```
    <provider
      android:name="FileProvider"
      android:authorities="com.commonsware.android.cp.files"
      android:exported="false"
      android:grantUriPermissions="false"
      tools:ignore="ExportedContentProvider">
      <grant-uri-permission android:path="/test.pdf"/>
    </provider>
```

With no other changes, if we tried to use the app, the third-party PDF viewer would crash when trying to read our PDF file from the Uri.

Step #2: Grant Access to the Uri

To allow third parties to get access only when we specify, we need to make a few more changes.

This <provider> element also has android:grantUriPermissions="false". That is the default value for this attribute, shown here purely for illustration purposes. It also has a <grant-uri-permissions> child element, listing the local path (within the ContentProvider) to our PDF file.

The <grant-uri-permissions> element (or elements, plural) allow us to override the permission requirement for certain pieces of content, granting access to that content on a per-request basis. There are three possibilities:

1. If android:grantUriPermissions is true, then we will be able to grant access to any content within our provider

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. If `android:grantUriPermissions` is `false`, but we have `<grant-uri-permissions>` sub-elements, we can only grant access to the content identified by the `Uri` paths specified in those sub-elements
3. If `android:grantUriPermissions` is `false`, and we have no `<grant-uri-permissions>` sub-elements (the default case), we cannot grant access to any content within our provider

In this case, we specify that we will only grant access to `/test.pdf`. Since that is the only content in this provider, we could have the same net effect by setting `android:grantUriPermissions` to `true`.

Then, when we create an `Intent` used to interact with another component, we can include a flag indicating what permission we wish to grant:

```
package com.commonsware.android.cp.perms;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;

public class FilesCPDemo extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    Intent i=new Intent(Intent.ACTION_VIEW, Uri.parse(FileProvider.CONTENT_URI +
"test.pdf"));

    i.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
    startActivity(i);
    finish();
  }
}
```

In this revised version of our activity, we add `FLAG_GRANT_READ_URI_PERMISSION` to the `Intent` used with `startActivity()`. This will grant the activity that responds to our `Intent` read access to the specific `Uri` in the `Intent`, overriding the exported status. That is why, when you run this app on a device, the PDF viewer will still be able to view the file.

There is also `FLAG_GRANT_WRITE_URI_PERMISSION` for granting write access, not needed here, as our provider only supports read access.

While this is most commonly used with `startActivity()` (e.g., allowing a mail program limited access to your attachments provider), this can also be used with

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`startService()`, `bindService()`, and the various flavors of sending broadcasts (e.g., `sendBroadcast()`).

The Stream Provider

Sometimes, we want a provider that looks like the local-file provider from the preceding section... but we do not have a file. Instead, we have data in some other form, such as a byte array, or a `String`, or an `InputStream`. Writing that material to a file may be problematic, or even counterproductive.

For example, imagine an app that stores data on the user's behalf in an encrypted fashion. One such file is a PDF, that the user would like to view. There are PDF viewers that can view files served via `content://` `Uri` values, as the previous section demonstrated... but that assumes an unencrypted file. While we could decrypt the file, writing the decrypted results to another file, and serve the decrypted data to the PDF viewer, now we have a persistent decrypted version of the data. That opens a window of time when the data might be accessed by people with nefarious intent, which is something we are trying to avoid by using the encrypted store in the first place. Rather, it would be nice if we could decrypt the data *on the fly* and give that decrypted result to the PDF viewer. Of course, there are security risks intrinsic to that too — after all, we do not know what the PDF viewer might do with the unencrypted data — but it is at least an improvement.

The good news is that Android does support streaming options for `openFile()`-style `ContentProvider` implementations. However, as one might expect, they are not the simplest things to implement.

In this section, we will examine the `ContentProvider/Pipe` sample project. This is a near clone of the `ContentProvider/Files` sample from the preceding section. However, rather than simply handing the file to Android to serve as content, we will stream it in ourselves. In principle, as part of this streaming, we could be decrypting it from an encrypted state. Since this sample shares much code with the previous sample, we will focus solely on the changes here.

Note that this sample was inspired by the sample found at https://github.com/nandeeshwar/Pfd-Create-Pipe.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Pipes

Starting with API Level 9, it is possible to create a pipe between two processes, from the Android SDK, via `ParcelFileDescriptor`. In the previous section, we saw how `ParcelFileDescriptor` could be used to open a local file and make that available to other processes — the `createPipe()` method gives us a pipe.

The "pipe" returned by `createPipe()` is a two-element array of `ParcelFileDescriptor` objects. The first element in the array represents the "read" end of the pipe. In our case, that is the end that should be used by a PDF viewer to read in the file contents. The second element of the array represents the "write" end of the pipe, which we will use to supply the file's contents to the "read" end (and to the PDF viewer by extension).

The Revised openFile()

With that in mind, here is our revised `openFile()` method:

```
  @Override
  public ParcelFileDescriptor openFile(Uri uri, String mode)
                                                       throws
FileNotFoundException {
    ParcelFileDescriptor[] pipe=null;

    try {
      pipe=ParcelFileDescriptor.createPipe();
      AssetManager assets=getContext().getResources().getAssets();

      new TransferThread(assets.open(uri.getLastPathSegment()),
                         new AutoCloseOutputStream(pipe[1])).start();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception opening pipe", e);
      throw new FileNotFoundException("Could not open pipe for: "
          + uri.toString());
    }

    return(pipe[0]);
  }
```

We create our pipe via `createPipe()`, then get an `InputStream` on our PDF file stored as an asset — unlike the `ContentProvider/Files` sample, we do not need to copy the asset to a local file now. We then kick off a background thread, implemented in an inner class named `TransferThread`, to actually copy the data from the asset to the write end of the pipe.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Rather than supply `TransferThread` with a `ParcelFileDescriptor` for the write end of the pipe, we supply an `OutputStream`. Specifically, we pass in a `ParcelFileDescriptor.AutoCloseOutputStream`. This is an `OutputStream` that knows to close the `ParcelFileDescriptor` when we close the stream. Otherwise, it behaves like a fairly typical `OutputStream`.

The Transfer

`TransferThread` is a fairly conventional copy-data-from-stream-to-stream implementation:

```
  static class TransferThread extends Thread {
    InputStream in;
    OutputStream out;

    TransferThread(InputStream in, OutputStream out) {
      this.in=in;
      this.out=out;
    }

    @Override
    public void run() {
      byte[] buf=new byte[1024];
      int len;

      try {
        while ((len=in.read(buf)) >= 0) {
          out.write(buf, 0, len);
        }

        in.close();
        out.flush();
        out.close();
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(),
              "Exception transferring file", e);
      }
    }
  }
```

Here, we read in data in 1KB blocks from the `InputStream` (our asset) and write the data to our `OutputStream` (obtained from the `ParcelFileDescriptor`).

The Results

Our activity logic has not substantially changed. We still create an `ACTION_VIEW` `Intent` on the `content://` `Uri` from our provider, pointing to our `test.pdf` asset.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Any PDF viewer capable of handling `content://` `Uri` values will use a `ContentResolver` to open an `InputStream` for our `Uri`. In the `ContentProvider/Files` sample, that `InputStream` would receive the contents of the file directly from Android. In this new sample, that `InputStream` is reading in bytes off of our pipe, until such time as it has read in all the streamed data and we have closed the `OutputStream`.

Not every possible consumer of a `Uri` will be able to work with our stream, though. For example, `MediaPlayer` expects to be able to determine, up front, how big the file is, and while that works for file-backed `ParcelFileDescriptors`, it does not work for those representing a pipe. Hence, `MediaPlayer` will crash when trying to use a `Uri` to a pipe-based stream, which is certainly unfortunate.

The author would like to thank Reuben Scratton for his assistance in tracking down this `MediaPlayer` limitation.

FileProvider

The Android Support package now contains its own implementation of a `FileProvider` that greatly simplifies serving files from internal or external storage to another app.

Here, we will see Google's `FileProvider` in action via the `ContentProvider/V4FileProvider` sample project. This is a near clone of the `ContentProvider/Pipe` sample from the preceding section, just without our own `ContentProvider` implementation, serving a file from internal storage.

The Rationale

The documentation for `FileProvider` states:

> Apps should generally avoid sending raw filesystem paths across process boundaries, since the receiving app may not have the same access as the sender. Instead, apps should send Uri backed by a provider like FileProvider.

This is not just an issue for passing files from internal storage to other apps. On Android 4.2+ tablets, it could even be an issue for external storage, as each user account gets its own portion of external storage. There may be scenarios in which your app (associated with one user) winds up needing to pass the contents of a file on external storage to another app (associated with another user). Regular filesystem

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

paths will not work in this case, as one user account cannot directly access another user account's files, even on external storage.

The Sources of Files

Google's `FileProvider` offers automatic serving of files from three root points:

- `getFilesDir()` (i.e., the standard portion of internal storage for your app)
- `getCacheDir()` (i.e., internal storage, but files that the OS can purge if needed to free up disk space)
- `Environment.getExternalStorageDirectory()` (i.e., the root of external storage)

For each of these, you will be able to specify a specific subdirectory's worth of files that should be served, if you do not want the entire directory's contents published via `FileProvider`. You will also be able to specify an alias, which serves as the first path segment (after the authority in the `content://` `Uri`) — `FileProvider` maps that path segment to a specific location of files to serve.

The Manifest Entry

The information about what files to serve comes in the form of an XML resource file. You can name the file whatever you like, but its content needs to be a root `<paths>` element, with a series of children for the different directories you wish to serve. Those directories will be denoted via child elements with specific names:

- `<files-path>` for `getFilesDir()`
- `<cache-path>` for `getCacheDir()`
- `<external-path>` for `Environment.getExternalStorageDirectory()`

For example, our sample project has a `res/xml/provider_paths.xml` file with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <files-path name="stuff" />
</paths>
```

Here, we are saying that we want to serve the contents of `getFilesDir()`, using a virtual root path of `stuff`. With an authority of `com.commonsware.android.cp.v4file`, this means that a `Uri` of

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`content://com.commonsware.android.cp.v4file/stuff/test.pdf` would serve up a `test.pdf` file in the `getFilesDir()` directory.

The optional `path` attribute of the `<files-path>`, etc. elements indicates a particular subdirectory, relative to the element-specific root, that should be used as the source of files. So, for example, had the `provider_paths.xml` file looked like:

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <files-path name="stuff" path="help/" />
</paths>
```

...then `content://com.commonsware.android.cp.v4file/stuff/test.pdf` would map to `help/test.pdf` inside of `getFilesDir()`.

You then point to this XML resource from a `<meta-data>` element in the `<provider>` element in the manifest, teaching `FileProvider` what to serve. For example, our `<provider>` element in this sample app is:

```
        android:resource="@xml/provider_paths"/>
    </provider>
  </application>

</manifest>
```

Here, our `android:name` points to Google's `FileProvider` (or, more accurately, `android.support.v4.content.FileProvider`), not some class of our own. We still provide the `android:authorities` value, along with any permission rules that we want. Beyond that, we have a `<meta-data>` element, with an `android:name` of `android.support.FILE_PROVIDER_PATHS`, that points to our XML resource with the path information.

You will also notice that our `android:exported` attribute is set to `false`. As it turns out, `FLAG_GRANT_READ_URI_PERMISSION` trumps the exported status of a provider. If you pass a `Uri` to an activity using `FLAG_GRANT_READ_URI_PERMISSION`, the activity will be able to read the contents of that `Uri`, even if the provider itself is not exported.

The Usage

At this point, the `FileProvider` is ready for use, insofar as we can specify `Uri` values like `content://com.commonsware.android.cp.v4file/stuff/test.pdf` and get

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

results. Of course, we actually need to have files in our internal storage, and we need to use such a `Uri`.

Hence, our activity combines the unpack-the-file-from-assets logic from our own providers in earlier samples, plus starts up a PDF viewer on our designated `test.pdf` file:

```
package com.commonsware.android.cp.v4file;

import android.app.Activity;
import android.content.Intent;
import android.content.res.AssetManager;
import android.os.Bundle;
import android.support.v4.content.FileProvider;
import android.util.Log;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;

public class FilesCPDemo extends Activity {
  private static final String AUTHORITY="com.commonsware.android.cp.v4file";

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    File f=new File(getFilesDir(), "test.pdf");

    if (!f.exists()) {
      AssetManager assets=getResources().getAssets();

      try {
        copy(assets.open("test.pdf"), f);
      }
      catch (IOException e) {
        Log.e("FileProvider", "Exception copying from assets", e);
      }
    }

    Intent i=
        new Intent(Intent.ACTION_VIEW,
                   FileProvider.getUriForFile(this, AUTHORITY, f));

    i.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
    startActivity(i);
    finish();
  }

  static private void copy(InputStream in, File dst) throws IOException {
    FileOutputStream out=new FileOutputStream(dst);
    byte[] buf=new byte[1024];
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    int len;

    while ((len=in.read(buf)) > 0) {
      out.write(buf, 0, len);
    }

    in.close();
    out.close();
  }
}
```

`FileProvider` offers a handy `getUriForFile()` static helper method that will return a `Uri` for a given file, incorporating our specified content provider authority.

The result of running this activity is the same as the other file-serving provider samples from this chapter: a PDF viewer (if one is available) will display the `test.pdf` file.

StreamProvider

`FileProvider` is rather nice: you can serve up typical file-based content without having to roll your own implementation of `ContentProvider` and `openFile()`. However, it only supports a few sources of data: `getFilesDir()`, `getCacheDir()`, and `Environment.getExternalStoragePublicDirectory()`.

The author of this book has written `StreamProvider`, a fork of `FileProvider` that adds support for serving content from assets, raw resources, `getExternalFilesDir()`, and `getExternalCacheDir()`. `StreamProvider` can be found in the CWAC-Provider project.

Once you have added the CWAC-Provider dependency to your project (e.g., added the JAR to `libs/`), you use it much the same as you would use `FileProvider`:

- Define an XML metadata file with a `<paths>` root element, containing one or more elements describing what you want the provider to serve
- Add `com.commonsware.cwac.provider.StreamProvider` as a `<provider>` to your manifest, under your own `android:authority`, with a `<meta-data>` element (with a name of `com.commonsware.cwac.provider.STREAM_PROVIDER_PATHS`), pointing to that XML metadata
- Use `FLAG_GRANT_READ_URI_PERMISSION` and `FLAG_GRANT_WRITE_URI_PERMISSION` in `Intent` objects you use to have third

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

parties use the files the `StreamProvider` serves, to allow those apps selective, temporary access to the file

The XML metadata can have:

- `<external-files-path>` for serving files from `getExternalFilesDir(null)`
- `<external-cache-path>` for serving files from `getExternalCacheDir()`
- `<raw-resource>` for serving a particular raw resource, where the path is the name of the raw resource (without file extension)
- `<asset>` for serving files from `assets/`

These are in addition to the `<files-path>`, `<external-path>`, and `<cache-path>` supported by `FileProvider`.

Hence, `StreamProvider` is especially useful when you want to package some content — such as a PDF file for online help — that you want to serve from your app. Just drop the file in `assets/` in your project, set up `StreamProvider` to serve up assets, and use an appropriate `Intent` with `startActivity()` to view that file.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Loader Framework

One perpetual problem in Android development is getting work to run outside the main application thread. Every millisecond we spend on the main application thread is a millisecond that our UI is frozen and unresponsive. Disk I/O, in particular, is a common source of such slowdowns, particularly since this is one place where the emulator typically out-performs actual devices. While disk operations rarely get to the level of causing an "application not responding" (ANR) dialog to appear, they can make a UI "janky".

Android 3.0 introduced a new framework to help deal with loading bulk data off of disk, called "loaders". The hope is that developers can use loaders to move database queries and similar operations into the background and off the main application thread. That being said, loaders themselves have issues, not the least of which is the fact that it is new to Android 3.0 and therefore presents some surmountable challenges for use in older Android devices.

This chapter will outline the programming pattern loaders are designed to solve, how to use loaders (both built-in and third-party ones) in your activities, and how to create your own loaders for scenarios not already covered.

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- database access
- content provider theory
- content provider implementations

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Cursors: Issues with Management

Android has had the concept of "managed cursors" since Android 1.0, and perhaps before that. A managed `Cursor` is one that an `Activity`... well... manages. More specifically:

1. When the activity is stopped, the managed `Cursor` is deactivated, freeing up all of the memory associated with the result set, and thereby reducing the activity's heap footprint while it is not in the foreground
2. When the activity is restarted, the managed `Cursor` is requeried, to bring back the deactivated data, along the way incorporating any changes in that data that may have occurred while the activity was off-screen
3. When the activity is destroyed, the managed `Cursor` is closed.

This is a delightful set of functionality. `Cursor` objects obtained from a `ContentProvider` via `managedQuery()` are automatically managed; a `Cursor` from `SQLiteDatabase` can be managed by `startManagingCursor()`.

The problem is that the `requery()` operation that is performed when the activity is restarted is executed on the main application thread. Many times, this is not a huge deal. However, given the nature of on-device flash and the Linux filesystem that many Android devices use (YAFFS2), it is entirely possible that what ordinarily is quick sometimes will not be. Also, you might be testing with small data sets, and your users might be working with bigger ones. As a result, the `requery()` may slow down your UI in ways that the user will notice.

Introducing the Loader Framework

The `Loader` framework was designed to solve three issues with the old managed `Cursor` implementation:

- Arranging for a `requery()` (or the equivalent) to be performed on a background thread)
- Arranging for the original query that populated the data in the first place to also be performed on a background thread, which the managed `Cursor` solution did not address at all
- Supporting loading things other than a `Cursor`, in case you have data from other sources (e.g., XML files, JSON files, Web service calls) that might be able to take advantage of the same capabilities as you can get from a `Cursor` via the loaders

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

There are three major pieces to the `Loader` framework: `LoaderManager`, `LoaderCallbacks`, and the `Loader` itself.

LoaderManager

`LoaderManager` is your gateway to the `Loader` framework. You obtain one by calling `getLoaderManager()` (or `getSupportLoaderManager()`, as is described later in this chapter). Via the `LoaderManager` you can initialize a `Loader`, restart that `Loader` (e.g., if you have a different query to use for loading the data), etc.

LoaderCallbacks

Much of your interaction with the `Loader`, though, comes from your `LoaderCallbacks` object, such as your activity if that is where you elect to implement the `LoaderCallbacks` interface. Here, you will implement three "lifecycle" methods for consuming a `Loader`:

1. `onCreateLoader()` is called when your activity requests that a `LoaderManager` initialize a `Loader`. Here, you will create the instance of the Loader itself, teaching it whatever it needs to know to go load your data
2. `onLoadFinished()` is called when the `Loader` has actually loaded the data — you can take those results and pour them into your UI, such as calling `swapCursor()` on a `CursorAdapter` to supply the fresh `Cursor`'s worth of data
3. `onLoaderReset()` is called when you should stop using the data supplied to you in the last `onLoadFinished()` call (e.g., the `Cursor` is going to be closed), so you can arrange to make that happen (e.g., call `swapCursor(null)` on a `CursorAdapter`)

When you implement the `LoaderCallbacks` interface, you will need to provide the data type of whatever it is that your `Loader` is loading (e.g., `LoaderCallbacks<Cursor>`). If you have several loaders returning different data types, you may wish to consider implementing `LoaderCallbacks` on multiple objects (e.g., instances of anonymous inner classes), so you can take advantage of the type safety offered by Java generics, rather than implementing `LoaderCallbacks<Object>` or something to that effect.

Loader

Then, of course, there is `Loader` itself.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Consumers of the Loader framework will use some concrete implementation of the abstract `Loader` class in their `LoaderCallbacks` `onCreateLoader()` method. API Level 11 introduced only one concrete implementation: `CursorLoader`, designed to perform queries on a `ContentProvider`, and described in a later section.

Honeycomb… Or Not

`Loader` and its related classes were introduced in Android 3.0 (API Level 11). If your application is only going to be deployed on such devices, you can use loaders "naturally" via the standard implementation.

If, however, you are interested in using loaders but also want to support pre-Honeycomb devices, the Android Support package offers its own implementation of `Loader` and the other classes. However, to use it, you will need to work within four constraints:

- You will need to add the Android Support package's JAR to your project (e.g., copy the JAR into your `libs/` directory and add it to your build path)
- You will need to inherit from `FragmentActivity`, not the OS base `Activity` class or other refinements (e.g., `MapActivity`), or from other classes that inherit from `FragmentActivity` (e.g., `SherlockFragmentActivity`).
- You will need to import the `support.v4` versions of various classes (e.g., `android.support.v4.app.LoaderManager` instead of `android.app.LoaderManager`)
- You will need to get your `LoaderManager` by calling `getSupportLoaderManager()`, instead of `getLoaderManager()`, on your `FragmentActivity`

These limitations are the same ones that you will encounter when using fragments on older devices. Hence, while loaders and fragments are not really related, you may find yourself adopting both of them at the same time, as part of incorporating the Android Support package into your project.

Using CursorLoader

Let's start off by examining the simplest case: using a `CursorLoader` to asynchronously populate and update a `Cursor` retrieved from a `ContentProvider`. This is illustrated in the `Loaders/ConstantsLoader` sample project, which is the same show-the-list-of-gravity-constants sample application that we examined previously, updated to use the `Loader` framework. Note that this project does not

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

use the Android Support package and therefore only supports API Level 11 and higher.

In `onCreate()`, rather than executing a `managedQuery()` to retrieve our constants, we ask our `LoaderManager` to initialize a loader, after setting up our `SimpleCursorAdapter` on a null `Cursor`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  adapter=new SimpleCursorAdapter(this,
                        R.layout.row, null,
                        new String[] {Provider.Constants.TITLE,
                                      Provider.Constants.VALUE},
                        new int[] {R.id.title, R.id.value});

  setListAdapter(adapter);
  registerForContextMenu(getListView());
  getLoaderManager().initLoader(0, null, this);
}
```

Using a `null` `Cursor` means we will have an empty list at the outset, a problem we will rectify shortly.

The `initLoader()` call on `LoaderManager` (retrieved via `getLoaderManager()`) takes three parameters:

- A locally-unique identifier for this loader
- An optional `Bundle` of data to supply to the loader
- A `LoaderCallbacks` implementation to use for the results from this loader (here set to be the activity itself, as it implements the `LoaderManager.LoaderCallbacks<Cursor>` interface)

The first time you call this for a given identifier, your `onCreateLoader()` method of the `LoaderCallbacks` will be called. Here, you need to initialize the `Loader` to use for this identifier. You are passed the identifier plus the `Bundle` (if any was supplied). In our case, we want to use a `CursorLoader`:

```
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
  return(new CursorLoader(this, Provider.Constants.CONTENT_URI,
                          PROJECTION, null, null, null));
}
```

`CursorLoader` takes a `Context` plus all of the parameters you would ordinarily use with `managedQuery()`, such as the content provider `Uri`. Hence, converting existing

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

code to use CursorLoader means converting your managedQuery() call into an invocation of the CursorLoader constructor inside of your onCreateLoader() method.

At this point, the CursorLoader will query the content provider, but do so on a background thread, so the main application thread is not tied up. When the Cursor has been retrieved, it is supplied to your onLoadFinished() method of your LoaderCallbacks:

```
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
  adapter.swapCursor(cursor);
}
```

Here, we call the new swapCursor() available on CursorAdapter, to replace the original null Cursor with the newly-loaded Cursor.

Your onLoadFinished() method will also be called whenever the data represented by your Uri changes. That is because the CursorLoader is registering a ContentObserver, so it will find out about data changes and will automatically requery the Cursor and supply you with the updated data.

Eventually, onLoaderReset() will be called. You are passed a Cursor object that you were supplied previously in onLoadFinished(). You need to make sure that you are no longer using that Cursor at this point — in our case, we swap null back into our CursorAdapter:

```
public void onLoaderReset(Loader<Cursor> loader) {
  adapter.swapCursor(null);
}
```

And that's pretty much it, at least for using CursorLoader. Of course, you need a content provider to make this work, and creating a content provider involves a bit of work.

What Else Is Missing?

The Loader framework does an excellent job of handling queries in the background. What it does not do is help us with anything else that is supposed to be in the background, such as inserts, updates, deletes, or creating/upgrading the database. It is all too easy to put those on the main application thread and therefore possibly encounter issues. Moreover, since the thread(s) used by the Loader framework are an

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

implementation detail, we cannot use those threads ourselves necessarily for the other CRUD operations.

Issues, Issues, Issues

Unfortunately, not all is rosy with the `Loader` framework.

There appears to be a bug in the Android Support package's implementation of the framework. If you use a `Loader` from a fragment that has `setRetainInstance()` set to `true`, you will not be able to use the `Loader` again after a configuration change, such as a screen rotation. This bug is not seen with the native API Level 11+ implementation of the framework.

Loaders Beyond Cursors

Loaders are not limited to loading something represented by a `Cursor`. You can load any sort of content that might take longer to load than you would want to spend on the main application thread. While the only concrete `Loader` implementation supplied by Android at this time loads a `Cursor` from a `ContentProvider`, you can create your own non-`Cursor` `Loader` implementation or employ one written by a third party.

What Happens When…?

Here are some other common development scenarios and how the `Loader` framework addresses them.

… the Data Behind the Loader Changes?

According to the `Loader` documentation, "They monitor the source of their data and deliver new results when the content changes".

The documentation is incorrect.

A `Loader` *can* "monitor the source of their data and deliver new results when the content changes". There is nothing in the framework that requires this behavior. Moreover, there are some cases where it is clearly a bad idea to do this — imagine a `Loader` loading data off of the Internet, needing to constantly poll some server to look for changes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The documentation for a Loader implementation should tell you the rules. Android's built-in `CursorLoader` does deliver new results, by means of a behind-the-scenes `ContentObserver`. `SQLiteCursorLoader` does not deliver new results at this time. `SharedPreferencesLoader` hands you a `SharedPreferences` object, which intrinsically is aware of any changes, and so `SharedPreferencesLoader` does nothing special here.

… the Configuration Changes?

The managed `Cursor` system that the `Loader` framework replaces would automatically `requery()` any managed `Cursor` objects when an activity was restarted. This would update the `Cursor` in place with fresh data after a configuration change. Of course, it would do that on the main application thread, which was not ideal.

Your Loader objects are retained across the configuration change automatically. Barring bugs in a specific `Loader` implementation, your Loader should then hand the new activity instance the data that was retrieved on behalf of the old activity instance (e.g., the `Cursor`).

Hence, you do not have to do anything special for configuration changes.

… the Activity is Destroyed?

Another thing the managed `Cursor` system gave you was the automatic closing of your `Cursor` when the activity was destroyed. The `Loader` framework does this as well, by triggering a reset of the `Loader`, which obligates the `Loader` to release any loaded data.

… the Activity is Stopped?

The final major feature of the managed `Cursor` system was that it would `deactivate()` a managed `Cursor` when the activity was stopped. This would release all of the heap space held by that `Cursor` while it was not on the screen. Since the `Cursor` was refreshed as part of restarting the activity, this usually worked fairly well and would help minimize pressure on the heap.

Alas, this does not appear to be supported by the `Loader` framework. The `Loader` is reset when an activity is destroyed, not stopped. Hence, the `Loader` data will continue to tie up heap space even while the activity is not in the foreground.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For many activities, this should not pose a problem, as the heap space consumed by their `Cursor` objects is modest. If you have an activity with a massive `Cursor`, though, you may wish to consider what steps you can take on your own, outside of the `Loader` framework, to help with this.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The ContactsContract Provider

One of the more popular stores of data on your average Android device is the contact list. This is particularly true with Android 2.0 and newer versions, which track contacts across multiple different "accounts", or sources of contacts. Some may come from your Google account, while others might come from Exchange or other services.

This chapter will walk you through some of the basics for accessing the contacts on the device. Along the way, we will revisit and expand upon our knowledge of using a `ContentProvider`.

First, we will review the contacts APIs, past and present. We will then demonstrate how you can connect to the contacts engine to let users pick and view contacts... all without your application needing to know much of how contacts work. We will then show how you can query the contacts provider to obtain contacts and some of their details, like email addresses and phone numbers. We wrap by showing how you can invoke a built-in activity to let the user add a new contact, possibly including some data supplied by your application.

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- content provider theory
- content provider implementations

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Introducing You to Your Contacts

Android makes contacts available to you via a complex `ContentProvider` framework, so you can access many facets of a contact's data — not just their name, but addresses, phone numbers, groups, etc. Working with the contacts `ContentProvider` set is simple... only if you have an established pattern to work with. Otherwise, it may prove somewhat daunting.

Organizational Structure

The contacts `ContentProvider` framework can be found as the set of `ContactsContract` classes and interfaces in the `android.provider` package. Unfortunately, there is a dizzying array of inner classes to `ContactsContract`.

Contacts can be broken down into two types: raw and aggregate. Raw contacts come from a sync provider or are hand-entered by a user. Aggregate contacts represent the sum of information about an individual culled from various raw contacts. For example, if your Exchange sync provider has a contact with an email address of `jdoe@foo.com`, and your Facebook sync provider has a contact with an email address of `jdoe@foo.com`, Android may recognize that those two raw contacts represent the same person and therefore combine those in the aggregate contact for the user. The classes relating to raw contacts usually have `Raw` somewhere in their name, and these normally would be used only by custom sync providers.

The `ContactsContract.Contacts` and `ContactsContract.Data` classes represent the "entry points" for the `ContentProvider`, allowing you to query and obtain information on a wide range of different pieces of information. What is retrievable from these can be found in the various `ContactsContract.CommonDataKinds` series of classes. We will see examples of these operations later in this chapter.

A Look Back at Android 1.6

Prior to Android 2.0, Android had no contact synchronization built in. As a result, all contacts were in one large pool, whether they were hand-entered by users or were added via third-party applications. The API used for this is the `Contacts` `ContentProvider`.

In principle, the `Contacts` `ContentProvider` should still work, as it is merely deprecated in Android 2.0.1, not removed. In practice, it has one big limitation: it

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

will only report contacts added directly to the device (as opposed to ones synchronized from Microsoft Exchange, Facebook, or other sources).

Pick a Peck of Pickled People

Let's start by finding a contact. After all, that's what the contacts system is for.

Contacts, like anything stored in a `ContentProvider`, is identified by a `Uri`. Hence, we need a `Uri` we can use in the short term, perhaps to read some data, or perhaps just to open up the contact detail activity for the user.

We could ask for a raw contact, or we could ask for an aggregate contact. Since most consumers of the contacts `ContentProvider` will want the aggregate contact, we will use that.

For example, take a look at the `Contacts/Pick` sample project, as this shows how to pick a contact from a collection of contacts, then display the contact detail activity. This application gives you a really big "Gimme!" button, which when clicked will launch the contact-selection logic:

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/pick"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:text="Gimme a contact!"
  android:layout_weight="1"
/>
```

Our first step is to determine the `Uri` to use to reference the collection of contacts we want to pick from. In the long term, there should be just one answer for aggregate contacts: `android.provider.ContactsContract.Contacts.People.CONTENT_URI`. However, that only works for Android 2.0 (SDK level 5) and higher. On older versions of Android, we need to stick with the original `android.provider.Contacts.CONTENT_URI`. To accomplish this, we will use a pinch of reflection to determine our `Uri` via a static initializer when our activity starts:

```
  private static Uri CONTENT_URI=null;

  static {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk>=5) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      try {
        Class<?>
clazz=Class.forName("android.provider.ContactsContract$Contacts");

        CONTENT_URI=(Uri)clazz.getField("CONTENT_URI").get(clazz);
      }
      catch (Throwable t) {
        Log.e("PickDemo", "Exception when determining CONTENT_URI", t);
      }
    }
    else {
      CONTENT_URI=Contacts.People.CONTENT_URI;
    }
  }
```

Then, you need to create an `Intent` for the `ACTION_PICK` on the chosen `Uri`, then start another activity (via `startActivityForResult()`) to allow the user to pick a piece of content of the specified type:

```
        Intent i=new Intent(Intent.ACTION_PICK, CONTENT_URI);

        startActivityForResult(i, PICK_REQUEST);
```

When that spawned activity completes with `RESULT_OK`, the `ACTION_VIEW` is invoked on the resulting contact `Uri`, as obtained from the `Intent` returned by the pick activity:

```
  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode==PICK_REQUEST) {
      if (resultCode==RESULT_OK) {
          startActivity(new Intent(Intent.ACTION_VIEW,
                                  data.getData()));
      }
    }
  }
```

The result: the user chooses a collection, picks a piece of content, and views it.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 586: The PickDemo sample application, as initially launched

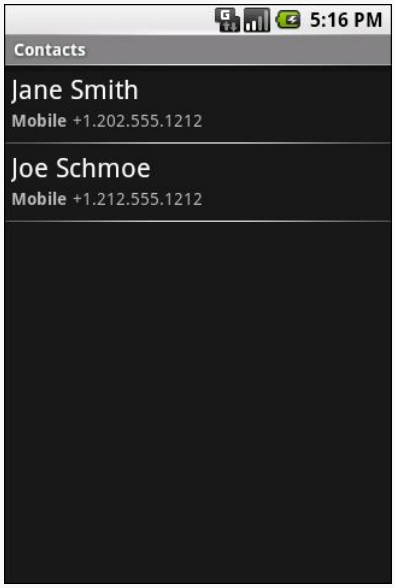


Figure 587: The same application, after clicking the "Gimme!" button, showing the list of available people

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

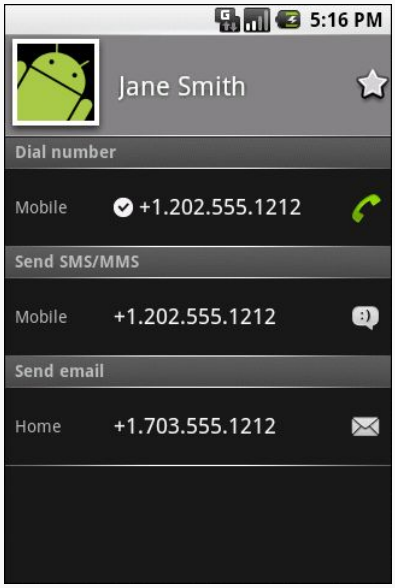


Figure 588: A view of a contact, launched by PickDemo after choosing one of the people from the pick list

Note that the `Uri` we get from picking the contact is valid in the short term, but should not be held onto in a persistent fashion (e.g., put in a database). If you need to try to store a reference to a contact for the long term, you will need to get a "lookup `Uri`" on it, to help deal with the fact that the aggregate contact may shift over time as raw contact information for that person comes and goes.

Spin Through Your Contacts

The preceding example allows you to work with contacts, yet not actually have any contact data other than a transient `Uri`. All else being equal, it is best to use the contacts system this way, as it means you do not need any extra permissions that might raise privacy issues.

Of course, all else is rarely equal.

Your alternative, therefore, is to execute queries against the contacts `ContentProvider` to get actual contact detail data back, such as names, phone numbers, and email addresses. The `Contacts/Spinners` sample application will demonstrate this technique.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Contact Permissions

Since contacts are privileged data, you need certain permissions to work with them. Specifically, you need the `READ_CONTACTS` permission to query and examine the `ContactsContract` content and `WRITE_CONTACTS` to add, modify, or remove contacts from the system. This only holds true if your code will have access to personally-identifying information, which is why the `Pick` sample above — which just has an opaque `Uri` — does not need any permission.

For example, here is the manifest for the `Contacts/Spinners` sample application:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.contacts.spinners"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-permission android:name="android.permission.READ_CONTACTS" />
  <uses-sdk android:minSdkVersion="3"
            android:targetSdkVersion="6" />
  <supports-screens android:largeScreens="true"
                    android:normalScreens="true"
                    android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name=".ContactSpinners">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Pre-Joined Data

While the database underlying the `ContactsContract` content provider is private, one can imagine that it has several tables: one for people, one for their phone numbers, one for their email addresses, etc. These are tied together by typical database relations, most likely 1:N, so the phone number and email address tables would have a foreign key pointing back to the table containing information about people.

To simplify accessing all of this through the content provider interface, Android pre-joins queries against some of the tables. For example, you can query for phone

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

numbers and get the contact name and other data along with the number — you do not have to do this join operation yourself.

The Sample Activity

The `ContactSpinners` activity is simply a `ListActivity`, though it sports a `Spinner` to go along with the obligatory `ListView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <Spinner android:id="@+id/spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:drawSelectorOnTop="false"
  />
</LinearLayout>
```

The activity itself sets up a listener on the `Spinner` and toggles the list of information shown in the `ListView` when the `Spinner` value changes:

```
package com.commonsware.android.contacts.spinners;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;
import android.widget.Spinner;

public class ContactSpinners extends ListActivity
  implements AdapterView.OnItemSelectedListener {
  private static String[] options={"Contact Names",
                                   "Contact Names & Numbers",
                                   "Contact Names & Email Addresses"};
  private ListAdapter[] listAdapters=new ListAdapter[3];

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    setContentView(R.layout.main);

    initListAdapters();

    Spinner spin=(Spinner)findViewById(R.id.spinner);
    spin.setOnItemSelectedListener(this);

    ArrayAdapter<String> aa=new ArrayAdapter<String>(this,
                              android.R.layout.simple_spinner_item,
                              options);

    aa.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);
    spin.setAdapter(aa);
  }

  public void onItemSelected(AdapterView<?> parent,
                             View v, int position, long id) {
    setListAdapter(listAdapters[position]);
  }

  public void onNothingSelected(AdapterView<?> parent) {
    // ignore
  }

  private void initListAdapters() {
    listAdapters[0]=ContactsAdapterBridge.INSTANCE.buildNameAdapter(this);
    listAdapters[1]=ContactsAdapterBridge.INSTANCE.buildPhonesAdapter(this);
    listAdapters[2]=ContactsAdapterBridge.INSTANCE.buildEmailAdapter(this);
  }

}
```

When the activity is first opened, it sets up three `Adapter` objects, one for each of three perspectives on the contacts data. The `Spinner` simply resets the list to use the `Adapter` associated with the `Spinner` value selected.

Dealing with API Versions

Of course, once again, we have to ponder different API levels.

Querying `ContactsContract` and querying `Contacts` is similar, yet different, both in terms of the `Uri` each uses for the query and in terms of the available column names for the resulting projection.

Rather than using reflection, this time we ruthlessly exploit a feature of the VM: classes are only loaded when first referenced. Hence, we can have a class that refers to new APIs (`ContactsContract`) on a device that lacks those APIs, so long as we do not reference that class.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To accomplish this, we define an abstract base class, `ContactsAdapterBridge`, that will have a singleton instance capable of running our queries and building a `ListAdapter` for each. Then, we create two concrete subclasses, one for the old API:

```
package com.commonsware.android.contacts.spinners;

import android.app.Activity;
import android.database.Cursor;
import android.provider.Contacts;
import android.widget.ListAdapter;
import android.widget.SimpleCursorAdapter;

class OldContactsAdapterBridge extends ContactsAdapterBridge {
  ListAdapter buildNameAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts.People._ID,
                                        Contacts.PeopleColumns.NAME
                                      };
    Cursor c=a.managedQuery(Contacts.People.CONTENT_URI,
                            PROJECTION, null, null,
                            Contacts.People.DEFAULT_SORT_ORDER);

    return(new SimpleCursorAdapter( a,
                                    android.R.layout.simple_list_item_1,
                                    c,
                                    new String[] {
                                      Contacts.PeopleColumns.NAME
                                    },
                                    new int[] {
                                      android.R.id.text1
                                    }));
  }

  ListAdapter buildPhonesAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts.Phones._ID,
                                        Contacts.Phones.NAME,
                                        Contacts.Phones.NUMBER
                                      };
    Cursor c=a.managedQuery(Contacts.Phones.CONTENT_URI,
                            PROJECTION, null, null,
                            Contacts.Phones.DEFAULT_SORT_ORDER);

    return(new SimpleCursorAdapter( a,
                                    android.R.layout.simple_list_item_2,
                                    c,
                                    new String[] {
                                      Contacts.Phones.NAME,
                                      Contacts.Phones.NUMBER
                                    },
                                    new int[] {
                                      android.R.id.text1,
                                      android.R.id.text2
                                    }));
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  ListAdapter buildEmailAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts.ContactMethods._ID,
                                        Contacts.ContactMethods.DATA,
                                        Contacts.PeopleColumns.NAME
                                      };
    Cursor c=a.managedQuery(Contacts.ContactMethods.CONTENT_EMAIL_URI,
                            PROJECTION, null, null,
                            Contacts.ContactMethods.DEFAULT_SORT_ORDER);

    return(new SimpleCursorAdapter( a,
                                    android.R.layout.simple_list_item_2,
                                    c,
                                    new String[] {
                                      Contacts.PeopleColumns.NAME,
                                      Contacts.ContactMethods.DATA
                                    },
                                    new int[] {
                                      android.R.id.text1,
                                      android.R.id.text2
                                    }));
  }
}
```

... and one for the new API:

```
package com.commonsware.android.contacts.spinners;

import android.app.Activity;
import android.database.Cursor;
import android.provider.ContactsContract.Contacts;
import android.provider.ContactsContract.CommonDataKinds.Email;
import android.provider.ContactsContract.CommonDataKinds.Phone;
import android.widget.ListAdapter;
import android.widget.SimpleCursorAdapter;

class NewContactsAdapterBridge extends ContactsAdapterBridge {
  ListAdapter buildNameAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts._ID,
                                        Contacts.DISPLAY_NAME,
                                      };
    Cursor c=a.managedQuery(Contacts.CONTENT_URI,
                            PROJECTION, null, null, null);

    return(new SimpleCursorAdapter( a,
                                    android.R.layout.simple_list_item_1,
                                    c,
                                    new String[] {
                                      Contacts.DISPLAY_NAME
                                    },
                                    new int[] {
                                      android.R.id.text1
                                    }));
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }

  ListAdapter buildPhonesAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts._ID,
                                        Contacts.DISPLAY_NAME,
                                        Phone.NUMBER
                                      };
    Cursor c=a.managedQuery(Phone.CONTENT_URI,
                            PROJECTION, null, null, null);

    return(new SimpleCursorAdapter( a,
                                    android.R.layout.simple_list_item_2,
                                    c,
                                    new String[] {
                                      Contacts.DISPLAY_NAME,
                                      Phone.NUMBER
                                    },
                                    new int[] {
                                      android.R.id.text1,
                                      android.R.id.text2
                                    }));
  }

  ListAdapter buildEmailAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts._ID,
                                        Contacts.DISPLAY_NAME,
                                        Email.DATA
                                      };
    Cursor c=a.managedQuery(Email.CONTENT_URI,
                            PROJECTION, null, null, null);

    return(new SimpleCursorAdapter( a,
                                    android.R.layout.simple_list_item_2,
                                    c,
                                    new String[] {
                                      Contacts.DISPLAY_NAME,
                                      Email.DATA
                                    },
                                    new int[] {
                                      android.R.id.text1,
                                      android.R.id.text2
                                    }));
  }
}
```

Our `ContactsAdapterBridge` class then uses the SDK level to determine which of those two classes to use as the singleton:

```
package com.commonsware.android.contacts.spinners;

import android.app.Activity;
import android.os.Build;
import android.widget.ListAdapter;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
abstract class ContactsAdapterBridge {
  abstract ListAdapter buildNameAdapter(Activity a);
  abstract ListAdapter buildPhonesAdapter(Activity a);
  abstract ListAdapter buildEmailAdapter(Activity a);

  public static final ContactsAdapterBridge INSTANCE=buildBridge();

  private static ContactsAdapterBridge buildBridge() {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk<5) {
      return(new OldContactsAdapterBridge());
    }

    return(new NewContactsAdapterBridge());
  }
}
```

Accessing Contact Information

The first `Adapter` shows the names of all of the contacts. Since all the information we seek is in the contact itself, we can use the `CONTENT_URI` provider, retrieve all of the contacts in the default sort order, and pour them into a `SimpleCursorAdapter` set up to show each person on its own row:

Assuming you have some contacts in the database, they will appear when you first open the `ContactSpinners` activity, since that is the default perspective:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

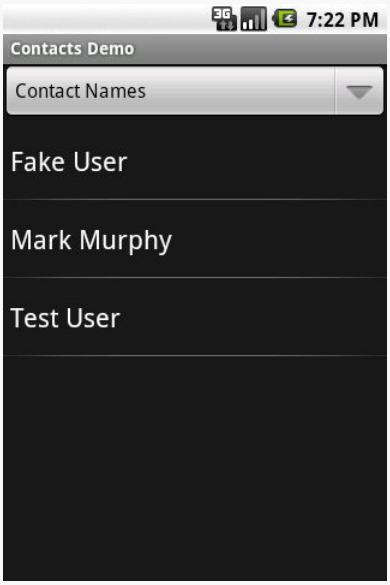


Figure 589: The ContactsDemo sample application, showing all contacts

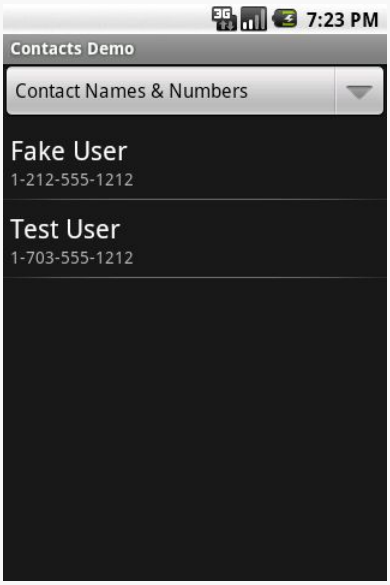


Figure 590: The ContactsDemo sample application, showing all contacts that have phone numbers

Similarly, to get a list of all the email addresses, we can use the `CONTENT_URI` content provider. Again, the results are displayed via a two-line `SimpleCursorAdapter`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 591: The ContactsDemo sample application, showing all contacts with email addresses

Makin' Contacts

Let's now take a peek at the reverse direction: adding contacts to the system. This was never particularly easy and now is... well, different.

First, we need to distinguish between sync providers and other apps. Sync providers are the guts underpinning the accounts system in Android, bridging some existing source of contact data to the Android device. Hence, you can have sync providers for Exchange, Facebook, and so forth. These will need to create raw contacts for newly-added contacts to their backing stores that are being sync'd to the device for the first time. Creating sync providers is outside of the scope of this book for now.

It is possible for other applications to create contacts. These, by definition, will be phone-only contacts, lacking any associated account, no different than if the user added the contact directly. The recommended approach to doing this is to collect the data you want, then spawn an activity to let the user add the contact — this avoids your application needing the `WRITE_CONTACTS` permission and all the privacy/data integrity issues that creates. In this case, we will stick with the new `ContactsContract` content provider, to simplify our code, at the expense of requiring Android 2.0 or newer.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To that end, take a look at the Contacts/Inserter sample project. It defines a simple activity with a two-field UI, with one field apiece for the person's first name and phone number:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="1"
  >
  <TableRow>
    <TextView
      android:text="First name:"
    />
    <EditText android:id="@+id/name"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Phone:"
    />
    <EditText android:id="@+id/phone"
      android:inputType="phone"
    />
  </TableRow>
  <Button android:id="@+id/insert" android:text="Insert!" />
</TableLayout>
```

The trivial UI also sports a button to add the contact:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

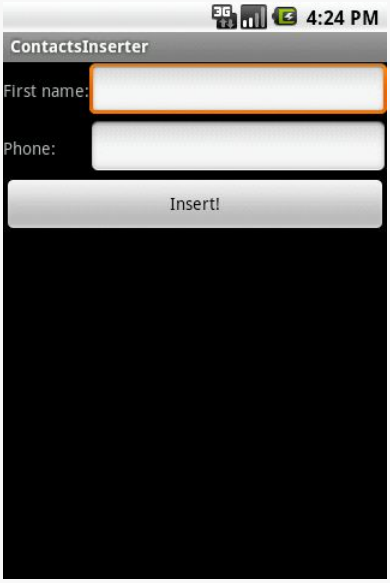


Figure 592: The ContactInserter sample application

When the user clicks the button, the activity gets the data and creates an `Intent` to be used to launch the add-a-contact activity. This uses the `ACTION_INSERT_OR_EDIT` action and a couple of extras from the `ContactsContract.Intents.Insert` class:

```
package com.commonsware.android.inserter;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.provider.ContactsContract.Contacts;
import android.provider.ContactsContract.Intents.Insert;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class ContactsInserter extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Button btn=(Button)findViewById(R.id.insert);

    btn.setOnClickListener(onInsert);
  }

  View.OnClickListener onInsert=new View.OnClickListener() {
    public void onClick(View v) {
      EditText fld=(EditText)findViewById(R.id.name);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      String name=fld.getText().toString();

      fld=(EditText)findViewById(R.id.phone);

      String phone=fld.getText().toString();
      Intent i=new Intent(Intent.ACTION_INSERT_OR_EDIT);

      i.setType(Contacts.CONTENT_ITEM_TYPE);
      i.putExtra(Insert.NAME, name);
      i.putExtra(Insert.PHONE, phone);
      startActivity(i);
    }
  };
}
```

We also need to set the MIME type on the `Intent` via `setType()`, to be `CONTENT_ITEM_TYPE`, so Android knows what sort of data we want to actually insert. Then, we call `startActivity()` on the resulting `Intent`. That brings up an add-or-edit activity:

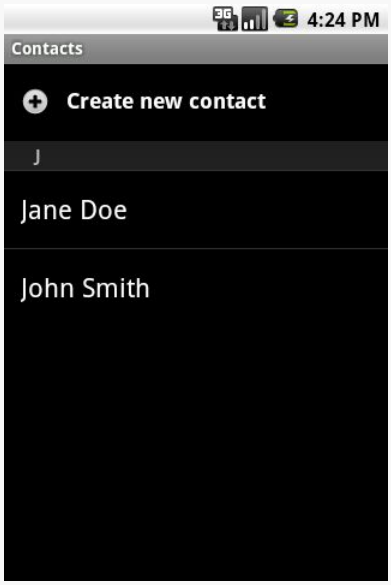


Figure 593: The add-or-edit-a-contact activity

... where if the user chooses "Create new contact", they are taken to the ordinary add-a-contact activity, with our data pre-filled in:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 594: The edit-contact form, showing the data from the ContactInserter activity

Note that the user could choose an existing contact, rather than creating a new contact. If they choose an existing contact, the first name of that contact will be overwritten with the data supplied by the `ContactsInserter` activity, and a new phone number will be added from those `Intent` extras.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The CalendarContract Provider

The Android Open Source Project (AOSP) has had a Calendar application from its earliest days. This application originally was designed to sync with Google Calendar, later extended to other sync sources, such as Microsoft's Exchange. However, this application was not part of the Android SDK, so there was no way to access it from your Android application.

At least, no officially documented and supported way.

Many developers poked through the AOSP source code and found that the Calendar application had a `ContentProvider`. Moreover, this `ContentProvider` was exported (by default). So many developers used undocumented and unsupported means for accessing calendar information. This occasionally broke, as Google modified the Calendar app and changed these pseudo-external interfaces.

Android 4.0 added official SDK support for interacting with the Calendar application via its `ContentProvider`. As part of the SDK, these new interfaces should be fairly stable — if nothing else, they should be supported indefinitely, even if new and improved interfaces are added sometime in the future. So, if you want to tie into the user's calendars, you can. Bear in mind, though, that the new `CalendarContract ContentProvider` is not identical to the older undocumented providers, so if you are aiming to support pre–4.0 devices, you have some more work to do.

Of course, similar to the `ContactsContract ContentProvider`, the `CalendarContract ContentProvider` is severely lacking in documentation, and anything not documented is subject to change.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- content provider theory
- content provider implementations

You Can't Be a Faker

While the Android 4.0 emulator has the `CalendarContract` `ContentProvider`, it will do you little good. While you can define a Google account on the emulator, the emulator lacks any ability to sync content with that account. Hence, you cannot see any events for your calendars in the Calendar app, and you cannot access any calendar data via `CalendarContract`.

Hence, at present, in order to test your use of `CalendarContract`, you will need to have hardware that runs Android 4.0 (or higher), with one or more accounts set up that have calendar data.

Do You Have Room on Your Calendar?

As a `ContentProvider`, `CalendarContract` is not significantly different from any other such provider that Android supplies or that you write yourself, in that there are `Uri` values representing collections of data, upon which you can query, insert, update, and delete as needed.

The Collections

The two main collections of data that you are likely to be interested in are `CalendarContract.Calendars` (the collection of all defined calendars) and `CalendarContract.Events` (the collection of all defined events across all calendars). Each of those has a `CONTENT_URI` static data member that you would use with `ContentResolver` or a `CursorLoader` to perform operations on those collections. An entry in `CalendarContract.Events` points back to its corresponding calendar via a `CALENDAR_ID` column that you can query upon; the remaining columns on `CalendarContract.Events` have names apparently designed to match with the iCalendar specification (e.g., `DTSTART` and `DTEND` for the start and end times of the event).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Three other collections may be of interest:

1. `CalendarContract.Instances` has one entry per occurrence of an event, so recurring events get multiple rows
2. `CalendarContract.Attendees` has information about each attendee of an event
3. `CalendarContract.Reminders` has information about each reminder scheduled for an event (e.g., when to remind the user), for those events with associated reminders

Each of those ties back to its associated `CalendarContract.Events` row via an `EVENT_ID` column.

Calendar Permissions

There are two permissions for working with `CalendarContract`: `READ_CALENDAR` and `WRITE_CALENDAR`. As you might expect, querying `CalendarContract` requires the `READ_CALENDAR` permission; modifying `CalendarContract` data requires the `WRITE_CALENDAR` permission.

These permissions have existed since Android's earliest days, even in the SDK, as a side effect of the "meat cleaver" approach the core Android team employed to create the initial SDK. Hence, you can request these permissions in the manifest with any Android build target, without compiler errors. Of course, actually referring to `CalendarContract` will require a build target of API Level 14 or higher.

Querying for Events

For example, let's populate a `ListView` with the roster of all events the user has across all calendars, using a `CursorLoader`, showing the name of each event, the event's start date, and the event's end date. You can find this in the `Calendar/Query` sample project in the book's source code.

Our manifest has the `READ_CALENDARS` permission, as you would expect:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.cal.query"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk android:minSdkVersion="14"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <uses-permission android:name="android.permission.READ_CALENDAR"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name=".CalendarQueryActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

We will use a simple `ListActivity` and so therefore do not need an activity layout. Our row layout (`res/layout/row.xml`) has three `TextView` widgets for the three pieces of data that we want to display:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/linearLayout1"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

  <TextView
    android:id="@+id/title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="4dip"
    android:layout_weight="1"
    android:ellipsize="end"
    android:textSize="20sp"/>

  <LinearLayout
    android:id="@+id/linearLayout2"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_marginRight="4dip"
    android:orientation="vertical">

    <TextView
      android:id="@+id/dtstart"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="top"
      android:textSize="10sp"/>

    <TextView
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      android:id="@+id/dtend"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="bottom"
      android:textSize="10sp"/>
  </LinearLayout>

</LinearLayout>
```

In our activity (`CalendarQueryActivity`), in `onCreate()`, we set up a `SimpleCursorAdapter` on a `null` `Cursor` at the outset and define the activity as being the adapter's `ViewBinder`:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    adapter=
        new SimpleCursorAdapter(this, R.layout.row, null, ROW_COLUMNS,
                                ROW_IDS);
    adapter.setViewBinder(this);
    setListAdapter(adapter);

    getLoaderManager().initLoader(0, null, this);
  }
```

A `ViewBinder` is a way to tailor how `Cursor` data is poured into row widgets, without subclassing the `SimpleCursorAdapter`. Implementing the `SimpleCursorAdapter.ViewBinder` interface requires us to implement a `setViewValue()` method, which will be called when the adapter wishes to pour data from one column of a `Cursor` into one widget. We will examine this method shortly.

The `SimpleCursorAdapter` will pour data from the `ROW_COLUMNS` in our `Cursor` into the `ROW_IDS` widgets in our row layout:

```
  private static final String[] ROW_COLUMNS=
      new String[] { CalendarContract.Events.TITLE,
          CalendarContract.Events.DTSTART,
          CalendarContract.Events.DTEND };
  private static final int[] ROW_IDS=
      new int[] { R.id.title, R.id.dtstart, R.id.dtend };
```

Our `onCreate()` also initializes the `Loader` framework, triggering a call to `onCreateLoader()`, where we create and return a `CursorLoader`:

```
  public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
    return(new CursorLoader(this, CalendarContract.Events.CONTENT_URI,
                            PROJECTION, null, null,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                              CalendarContract.Events.DTSTART));
  }
```

We query on `CalendarContract.Events.CONTENT_URI`, asking for a certain set of columns indicated by our `PROJECTION` static data member:

```
  private static final String[] PROJECTION=
      new String[] { CalendarContract.Events._ID,
          CalendarContract.Events.TITLE,
          CalendarContract.Events.DTSTART,
          CalendarContract.Events.DTEND };
```

The `ROW_COLUMNS` we map are a subset of the `PROJECTION`, skipping the `_ID` column that `SimpleCursorAdapter` needs but will not be displayed. Our query is also set up to sort by the start date (`CalendarContract.Events.DTSTART`).

When the query is complete, we pop it into the adapter in `onLoadFinished()` and remove it in `onLoaderReset()`:

```
  public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    adapter.swapCursor(cursor);
  }

  public void onLoaderReset(Loader<Cursor> loader) {
    adapter.swapCursor(null);
  }
```

Our `setViewValue()` implementation then converts the `DTSTART` and `DTEND` values into formatted strings by way of `DateUtils` and the `formatDateTime()` method:

```
  @Override
  public boolean setViewValue(View view, Cursor cursor, int columnIndex) {
    long time=0;
    String formattedTime=null;

    switch (columnIndex) {
      case 2:
      case 3:
        time=cursor.getLong(columnIndex);
        formattedTime=
            DateUtils.formatDateTime(this, time,
                                     DateUtils.FORMAT_ABBREV_RELATIVE);
        ((TextView)view).setText(formattedTime);
        break;

      default:
        return(false);
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(true);
  }
}
```

The `setViewValue()` method should return `true` for any columns it handles and `false` for columns it does not — skipped columns are handled by `SimpleCursorAdapter` itself.

If you run this on a device with available calendar data, you will get a list of those events:

Figure 595: The Calendar Query sample application, with some events redacted

Penciling In an Event

What is rarely documented in the Android SDK is what activities might exist that support the MIME types of a given `ContentProvider`. In part, that is because device manufacturers have the right to remove or replace many of the built-in applications.

The Calendar application is considered by Google to be a "core" application. Quoting the Android 2.3 version of the Compatibility Definition Document (CDD):

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

> The Android upstream project defines a number of core applications, such as a phone dialer, calendar, contacts book, music player, and so on. Device implementers MAY replace these applications with alternative versions. However, any such alternative versions MUST honor the same Intent patterns provided by the upstream project. For example, if a device contains an alternative music player, it must still honor the Intent pattern issued by third-party applications to pick a song.

Hence, in theory, so long as the CDD does not change and device manufacturers correctly honor it, those Intent patterns described by the Calendar application's manifest should be available across Android 4.0 devices. The Calendar application appears to support `ACTION_INSERT` and `ACTION_EDIT` for both the collection MIME type (`vnd.android.cursor.dir/event`) and the instance MIME type (`vnd.android.cursor.item/event`). Notably, there is no support for `ACTION_PICK` to pick a calendar or event, the way you can use `ACTION_PICK` to pick a contact.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The MediaStore Provider

Playing back media is a popular pastime on Android devices, one in which your app may want to participate. The easiest way for you to find out what media is available for you to display, edit, or otherwise work with is via the `MediaStore` content provider. `MediaStore` is part of the Android framework and allows you to query for images, audio files, and video files that are indexed on the device.

This chapter will review the general workings of `MediaStore`, plus work through an example of getting video files — and their thumbnails — from `MediaStore`.

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- content provider theory
- content provider implementations

It is also a pretty good idea to have read the chapters on media recording and playback that might be of relevance, depending on what you intend to do with the `MediaStore`:

- Audio Playback
- Audio Recording
- Video Playback
- Using the Camera via 3rd-Party Apps
- Working Directly with the Camera

You might also wish to consider skimming through the chapter on files again, as it will be cross-referenced in several places in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What Is the MediaStore?

The documentation for `MediaStore` describes it this way:

> The Media provider contains meta data for all available media on both internal and external storage devices.

This definition... leaves a bit to be desired.

From our standpoint as Android developers, the `MediaStore` is a `ContentProvider`, supplied by Android. We can use it much like we use other system-supplied providers, like `ContactsContract` and `CalendarContract`. In this case, the primary role of `MediaStore` is for us to find media, just as the primary role of `ContactsContract` is for us to find contacts.

The "meta data" reference in the documentation refers to the fact that `MediaStore` itself does not store the media, even though that's what the name `MediaStore` would suggest. `MediaMetadataStore` would be a more accurate description. We can learn *about* available media — names, durations, etc. — and we can get a `Uri` from `MediaStore` pointing to the media, but the media itself lives as a file somewhere else.

Indexed Media

`MediaStore` has media as a primary focus. Here, "media" refers to:

- Images (typically photos)
- Audio (music, podcasts, etc.)
- Video (whether recorded by the device, downloaded from somewhere, etc.)

`MediaStore` has intrinsic knowledge of these, particularly for the file formats and codecs that Android supports. As a result, the index maintained by `MediaStore` will contain some metadata in common for all file types, such as:

- title
- MIME type
- dates (when the file was added, when the file was modified)

...and other metadata that will be unique to one or two of the major types, such as:

- duration for audio and video (but not images)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- height and width for images and video (but not audio)
- geotagging for images and video (but not audio)

Indexed Non-Media

As was mentioned in passing in the chapter on files, Android uses MTP for Android 4.0+ as the USB protocol for sharing files with a desktop or notebook computer.

To power this, Android does not go straight to the filesystem, but rather works with `MediaStore`. `MediaStore` maintains an index of *all* files, not just "media". Whatever shows up in `MediaStore` is what shows up to the user in their Windows drive letter, OS X mounted volume, etc.

You too can query `MediaStore` for non-media files. Android will try to maintain a MIME type — probably based on file extensions — and so you can find all indexed PDF files, for example, by querying `MediaStore`.

MediaStore and "Other" External Storage

In the chapter on files, we covered the difference between internal storage and external storage. Primarily, `MediaStore` maintains an index of external storage.

However, many Android devices today have multiple locations that could be considered "external storage". While the vast majority of Android devices have "external storage" as a portion of on-board flash memory, Android device manufacturers are welcome to add other options, such as:

- card slots (typically microSD)
- USB host ports (capable of mounting thumb drives and the like)

From the standpoint of the Android SDK, such secondary storage locations are off-limits, in that there is nothing in the Android SDK to tell us if there are any such locations, where they are located (in terms of `File` objects to their roots), whether they can be read from, or whether they can be written to. You will find various blog posts and Stack Overflow answers where developers have attempted to catalog all of the possibilities, using a mix of low-level Linux information and manufacturer-based heuristics, but these techniques will be generally unreliable across thousands of device models.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, many manufacturers who have added such secondary storage options will arrange to have that storage be indexed and be part of `MediaStore`. So, if the user slides in a microSD card containing audio files, on many devices, when you query `MediaStore` for available audio files, you will find those on the microSD card in addition to those on "traditional" external storage. From the user's standpoint, in terms of consuming media, this is sufficient.

How Does My Content Get Indexed?

As was noted back in the chapter on files, if you write files to external storage, you will want to use `MediaScannerConnection` to ensure that those files get indexed. In that chapter, the focus was on ensuring that your files would be visible to attached desktops/notebooks via MTP. However, what *really* happens is that `MediaScannerConnection` updates `MediaStore`, which in turn drives the MTP-served content.

Even if you fail to index content manually, at some point, Android is likely to pick up the files. For example, Android will scan external storage after a reboot. However, using `MediaScannerConnection` to "tap Android on the shoulder" and have it index your file means that it will show up in `MediaStore` more quickly. This is very important for multimedia assets — if you downloaded some media, you want that to be indexed as soon as possible, so the user can turn around and consume that media, whether through your app or another one on the user's device.

How Do I Retrieve Video from the MediaStore?

Video players will need to find out what videos are available on the device, eligible for playback. They may wish to retrieve other details, such as the video title, duration, and so forth. And, of course, they will need something that they can use to actually play back the video itself.

In this section, we will work through the `Media/VideoList` sample project. This project has a `VideosFragment` that will show the roster of available videos; tapping on a video in the list will launch the user's video player to watch that video.

Requesting Permission

Starting on API Level 19 devices, you need to hold the `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` permissions to be able to work with the `MediaStore`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Hence, the VideoList sample app has the READ_EXTERNAL_STORAGE permission in its manifest, as it has no need to write to external storage:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.video.list"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"
    android:xlargeScreens="true"/>

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="17"/>

  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

  <application
    android:allowBackup="false"
    android:hardwareAccelerated="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Holo.Light.DarkActionBar">
    <activity
      android:name=".MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Querying for Video

VideosFragment uses the Loader framework, since MediaStore is a ContentProvider and Loader is a convenient way to asynchronously load content from a ContentProvider. VideosFragment implements the LoaderManager.LoaderCallbacks interface and, in onActivityCreated(), calls getLoaderManager().initLoader() to initialize its Loader.

That triggers a call to onCreateLoader(), where VideosFragment creates a CursorLoader to query the MediaStore for videos:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public Loader<Cursor> onCreateLoader(int arg0, Bundle arg1) {
    return(new CursorLoader(
                            getActivity(),
                            MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
                            null, null, null,
                            MediaStore.Video.Media.TITLE));
  }
```

The `Uri` for video content from `MediaStore` is `MediaStore.Video.Media.EXTERNAL_CONTENT_URI`. Passing in `null` for the list of columns to return will return all available columns — not the most efficient approach, but it is convenient. The sort order of `MediaStore.Video.Media.TITLE` has the results sorted by the `TITLE` column, so the videos are returned alphabetically.

Back up in `onActivityCreated()`, we initialized a `SimpleCursorAdapter` to handle our results, passing in the `TITLE` and `_ID` columns into our custom row layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal"
  android:padding="8dp">

  <ImageView
    android:id="@+id/thumbnail"
    android:layout_width="64dp"
    android:layout_height="64dp"
    android:contentDescription="@string/thumbnail"/>

  <TextView
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="8dp"
    android:layout_gravity="center_vertical"
    android:textSize="24sp"/>

</LinearLayout>
```

`onActivityCreated()` also attaches a custom `ViewBinder`, `ThumbnailBinder`, that we will cover in the next section, before eventually attaching the initially-empty `SimpleCursorAdapter` to the `ListView` of our `ListFragment`:

```
  @Override
  public void onActivityCreated(Bundle state) {
    super.onActivityCreated(state);

    String[] from=
        { MediaStore.Video.Media.TITLE, MediaStore.Video.Media._ID };
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    int[] to= { android.R.id.text1, R.id.thumbnail };
    SimpleCursorAdapter adapter=
        new SimpleCursorAdapter(getActivity(), R.layout.row, null,
                                from, to, 0);

    adapter.setViewBinder(this);
    setListAdapter(adapter);

    getLoaderManager().initLoader(0, null, this);
  }
```

The rest of our `LoaderManager.LoaderCallbacks` methods are fairly conventional, using `swapCursor()` to load in the results of the query (or `null` if the loader is reset):

```
  @Override
  public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
    ((CursorAdapter)getListAdapter()).swapCursor(c);
  }

  @Override
  public void onLoaderReset(Loader<Cursor> loader) {
    ((CursorAdapter)getListAdapter()).swapCursor(null);
  }
```

Showing the Thumbnails

If you have used a video player on Android, most have an activity (or fragment) akin to the one we are implementing in this section. And, most of those will show thumbnail images of the videos in question.

However, retrieving and showing those thumbnails is a bit complicated, because Android may need to *generate* the thumbnail, if there is not already a thumbnail for the video, or if its cache of thumbnails was cleared. Generating a thumbnail takes time, time that we do not want to spend on the main application thread.

So we need to load this image asynchronously. Picasso reportedly supports this, but at the time of this writing, it remains undocumented and hence may not be reliable.

Instead, this sample app will use another popular image loading library: the Universal Image Loader (UIL).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Preparing the ImageLoader

UIL works off of an `ImageLoader` to handle the asynchronous loading of images for some scope, such as for a fragment. Hence, our fragment holds an `ImageLoader` instance, which we configure in `onAttach()`, as we need an `Activity` to set up the loader:

```
  @Override
  public void onAttach(Activity host) {
    super.onAttach(host);

    ImageLoaderConfiguration ilConfig=
        new ImageLoaderConfiguration.Builder(getActivity()).build();

    imageLoader=ImageLoader.getInstance();
    imageLoader.init(ilConfig);
  }
```

UIL supports many configuration options for the `ImageLoader`, all of which we are ignoring here, settling for the defaults.

Attaching the ViewBinder

Next, the thumbnail will need to be displayed using some sort of `ImageView`. Since `SimpleCursorAdapter` cannot populate an `ImageView` directly, we need some other way to fill in the `ImageView`. To handle this, we create an implementation of a `ViewBinder`, named `ThumbnailBinder` — that is what we attached to our `SimpleCursorAdapter` via `setViewBinder()` back in `onActivityCreated()`.

A `ViewBinder` is a way to tailor how `Cursor` data is poured into row widgets, without subclassing the `SimpleCursorAdapter`. Implementing the `SimpleCursorAdapter.ViewBinder` interface requires us to implement a `setViewValue()` method, which will be called when the adapter wishes to pour data from one column of a `Cursor` into one widget:

```
  @Override
  public boolean setViewValue(View v, Cursor c, int column) {
    if (column == c.getColumnIndex(MediaStore.Video.Media._ID)) {
      Uri video=

ContentUris.withAppendedId(MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
              c.getInt(column));
      DisplayImageOptions opts=new DisplayImageOptions.Builder()
          .showImageOnLoading(R.drawable.ic_media_video_poster)
          .build();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      imageLoader.displayImage(video.toString(), (ImageView)v, opts);

      return(true);
    }

    return(false);
  }
```

If our `ViewBinder` is being asked to bind the video ID column (`MediaStore.Video.Media._ID`), we first construct the `Uri` to the video using `ContentUris.withAppendedId()`. Then, we create a UIL-supplied `DisplayImageOptions` object, where we can provide details for how to handle the image load. In this case, we supply a drawable resource to use as a placeholder image while the thumbnail is being loaded. Finally, we tell the `imageLoader` we set up in `onAttach()` to display the image. UIL recognizes the `Uri` structure for a video from `MediaStore` and uses utility methods like `getThumbnail()` on `MediaStore.Video.Thumbnails` to actually retrieve the thumbnail.

The net result is that when we populate our `ListView` with our `ViewBinder`-enhanced `SimpleCursorAdapter`, the `ListView` rows will initially have the placeholder image, replaced by the actual video thumbnails as they get loaded.

Playing the Selection

`VideosFragment` extends a version of `ContractListFragment`, as was used in the EU4You samples earlier in this book. The activity that hosts this fragment is obligated to implement the `VideosFragment.Contract` interface, which in turn requires an `onVideoSelected()` method.

In `onListItemClick()` of `VideosFragment`, the fragment calls `onVideoSelected()` on the `Contract`, supplying:

- the `String` representation of the `Uri` that points to the video itself, pulled from the `MediaStore.Video.Media.DATA` column of the `Cursor` we loaded from the `MediaStore`
- the MIME type of that video, pulled from the `MediaStore.Video.Media.MIME_TYPE` column of that same `Cursor`

```
public void onActivityCreated(Bundle state) {
  super.onActivityCreated(state);

  String[] from=
      { MediaStore.Video.Media.TITLE, MediaStore.Video.Media._ID };
  int[] to= { android.R.id.text1, R.id.thumbnail };
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    SimpleCursorAdapter adapter=
        new SimpleCursorAdapter(getActivity(), R.layout.row, null,
                                from, to, 0);

    adapter.setViewBinder(this);
```

The main activity — surprisingly named `MainActivity` — loads up a `VideosFragment` as a static fragment via the `res/layout/main.xml` resource:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/videos"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:name="com.commonsware.android.video.list.VideosFragment"
/>
```

`MainActivity` implements `VideosFragment.Contract` and therefore has an `onVideoSelected()` method. It simply constructs an `Intent` to view the video and starts an activity with it:

```
package com.commonsware.android.video.list;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import java.io.File;

public class MainActivity extends Activity implements
    VideosFragment.Contract {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  @Override
  public void onVideoSelected(String uri, String mimeType) {
    Uri video=Uri.fromFile(new File(uri));
    Intent i=new Intent(Intent.ACTION_VIEW);

    i.setDataAndType(video, mimeType);
    startActivity(i);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Results

Running this on a device with videos available should show the list of those videos, complete with title and thumbnail:

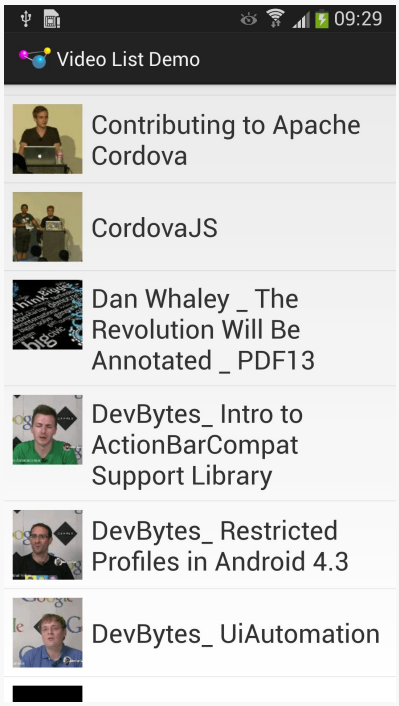


Figure 596: The Video List Demo App

Tapping on any entry in the list should bring up a video player on your device, assuming that one or more such players (that are capable of supporting `content://Uri` values) are installed.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Consuming Documents

Android has long offered the ability for an app to pick some file or stream from another app and consume it. However, the original options were designed around an app loading content from another app. Even though our *code* would be requesting content based on abstractions like MIME types, the *implementation* and user experience would be based on the traditional "pick an app to fulfill this request" chooser.

Google, given its clear interest in cross-cutting storage engines like Google Drive, wanted something better. In Android 4.4, they added the Storage Access Framework (SAF) to provide a better user experience, with only modest changes to client code.

In this chapter, we will examine what it takes to consume documents published via the SAF.

Prerequisites

This chapter assumes that you have read the chapter on `ContentProvider` patterns or have equivalent experience with consuming streams published by a `ContentProvider`.

The Storage Access… What?

Let's think about photos for a minute.

A person might have photos managed as:

- on-device photos, mediated by an app like a gallery

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- photos stored online in a photo-specific service, like Instagram
- photos stored online in a generic file-storage service, like Google Drive or Dropbox

Now, let's suppose that person is in an app that allows the user to pick a photo, such as to attach to an email.

The classic Android solution would be for the user to have to first choose the app to use to find the photo (e.g., Gallery, Instagram, Google Drive, Dropbox), then find the photo using that app. Then, if all goes well, the original app would receive a `Uri` to that photo and be able to make use of it.

However, this flow has three main problems:

1. From the user's standpoint, they need to know where they have the photo before they can go looking for it. Given the prominence of generic file-storage services, the user might not remember where the photo is stored, but might remember enough details about the photo (e.g., timeframe when taken, tags that might have been attached to the photo) to find it... but the user has to sequentially search each possible photo-storing app until the right one is found.
2. From the client app developer's standpoint, too many apps screw up handling the classic `ACTION_PICK` and `ACTION_GET_CONTENT` activities, failing to return a result in all cases. Users then are as likely to blame the client app for the mistake as they are to blame the photo-storing app, or Android itself.
3. None of this was designed with online file-sharing services in mind. What happens if an app knows about a possible file, but the file is not available on the device right now, because it has not been downloaded from the online service?

The Storage Access Framework is designed to address these issues. It provides its own "picker" UI to allow users to find a file of interest that matches the MIME type that the client app wants. File providers simply publish details about their available files — including those that may not be on the device but could be retrieved if needed. The picker UI allows for easy browsing and searching across all possible file providers, to streamline the process for the user. And, since Android is the one providing the picker, the picker should more reliably give a result to the client app based upon the user's selection (if any).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Storage Access Framework Participants

Providers are specialized `ContentProvider` implementations, usually extending `DocumentsProvider`, that can tell Android about the documents that are published by an app. This includes providing any sort of organizational structure (directory tree, tag cloud, etc.)

The **clients** are apps that wish to consume (or create) documents managed by providers. Clients will indicate what sort of document they want, in the form of a MIME type, where applicable.

The **picker** is the system UI that allows the user to pick a document (or documents) from among the documents published by all providers that meet the criteria established by a client requesting access to the document(s).

Picking How to Pick (a Peck of Pickled Pepper Photos)

`ACTION_PICK` would seem to be the `Intent` action to use to pick something. It works, but it is designed for the case where you know the specific collection of "somethings" you want to pick from. Use this, for example, to pick a contact specifically out of `ContactsContract`.

In cases where you know the MIME type you want, but you do not particularly know or care about the exact source of the file, use `ACTION_GET_CONTENT` on API Level 18 and below for everything.

For MIME types that clearly represent a document, file, or other sort of stream, use `ACTION_OPEN_DOCUMENT` (and the SAF) on API Level 19+. The SAF picker will incorporate both full-fledged SAF-compliant providers' documents along with apps that only support `ACTION_GET_CONTENT`. However, since `ACTION_OPEN_DOCUMENT` is only available on API Level 19+ devices, if you are supporting older devices, you will need to check `Build.VERSION.SDK_INT` and choose an `Intent` action accordingly.

For MIME types that represent entries in a database (e.g., a calendar entry), use `ACTION_GET_CONTENT`, even on API Level 19+. Google also recommends using `ACTION_GET_CONTENT` on API Level 19+ "if you want your app to simply read/import data", though it is unclear why they make this recommendation or why the user experience should differ based upon how the bytes would be used.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Opening a Document

Technically, we do not "open" a document using `ACTION_OPEN_DOCUMENT`. Instead, we are requesting a `Uri` pointing to some document that the user chooses.

To do that, create an `Intent` with:

- `ACTION_OPEN_DOCUMENT` as the action
- `CATEGORY_OPENABLE` as the category
- your desired MIME type

Then, use that `Intent` with `startActivityForResult()`.

For example, the Documents/Consumer sample application contains a `ConsumerFragment` that adds an "Open" item to the action bar overflow. Clicking on "Open" triggers a call to the `open()` method on the fragment. And, for API Level 19+ devices, that will in turn request to "open" a document:

```
  @TargetApi(Build.VERSION_CODES.KITKAT)
  private void open() {
    Intent i=new Intent().setType("image/*");

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
      startActivityForResult(i.setAction(Intent.ACTION_OPEN_DOCUMENT)
                              .addCategory(Intent.CATEGORY_OPENABLE),
                             REQUEST_OPEN);
    }
    else {
      startActivityForResult(i.setAction(Intent.ACTION_GET_CONTENT),
                             REQUEST_GET);
    }
  }
}
```

This `open()` method also gracefully degrades for older devices, falling back to `ACTION_GET_CONTENT`. In both cases, we are trying to allow the user to pick some image (MIME type of `image/*`). The two `startActivityForResult()` calls use different request IDs (`REQUEST_OPEN` versus `REQUEST_GET`), so that we can distinguish the sort of result that we get in `onActivityResult()`:

```
  @Override
  public void onActivityResult(int requestCode, int resultCode,
                               Intent resultData) {
    if (resultCode == Activity.RESULT_OK) {
      Uri uri=null;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (resultData != null) {
      uri=resultData.getData();
      logToTranscript(uri.toString());

      if (requestCode == REQUEST_OPEN) {
        Cursor c=
            getActivity().getContentResolver().query(uri, null, null,
                                                     null, null);

        if (c != null && c.moveToFirst()) {
          int displayNameColumn=
              c.getColumnIndex(OpenableColumns.DISPLAY_NAME);

          if (displayNameColumn >= 0) {
            logToTranscript("Display name: "
                + c.getString(displayNameColumn));
          }

          int sizeColumn=c.getColumnIndex(OpenableColumns.SIZE);

          if (sizeColumn < 0 || c.isNull(sizeColumn)) {
            logToTranscript("Size not available");
          }
          else {
            logToTranscript(String.format("Size: %d",
                                          c.getInt(sizeColumn)));
          }
        }
        else {
          logToTranscript("...no metadata available?");
        }
      }
    }
  }
}
```

Both `ACTION_GET_CONTENT` and `ACTION_OPEN_DOCUMENT` should supply a `Uri` in the result `Intent` that points to the document the user chose, if the user actually chose one and we got `RESULT_OK` as the result code. This sample logs that `Uri` value to a "transcript" (`TextView` inside of a `ScrollView`) to show what we get back.

If the result is from an `ACTION_OPEN_DOCUMENT` request (`REQUEST_OPEN` request code), we can try to get some metadata about the document. The provider should support a query on the returned `Uri` that will give us the display name (`OpenableColumns.DISPLAY_NAME`) and possibly the size of the file (`OpenableColumns.SIZE`). So, we use a `ContentResolver` to run this query, and if we get results back, we try to read out these two values and record them to the transcript as well. Note, though, that:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- There is no guarantee that either column will be in the result set, or that the result set will have any rows
- The size might not be known, particularly if the file is not presently resident on the device (e.g., it is being downloaded now, given that the user chose the file), and so we need to call `isNull()` on the `Cursor` to see if we actually have a `SIZE` value before trying to get it as an integer

The user is presented with the system's picker, to choose an image, complete with a navigation drawer to get to various spots within the picker:

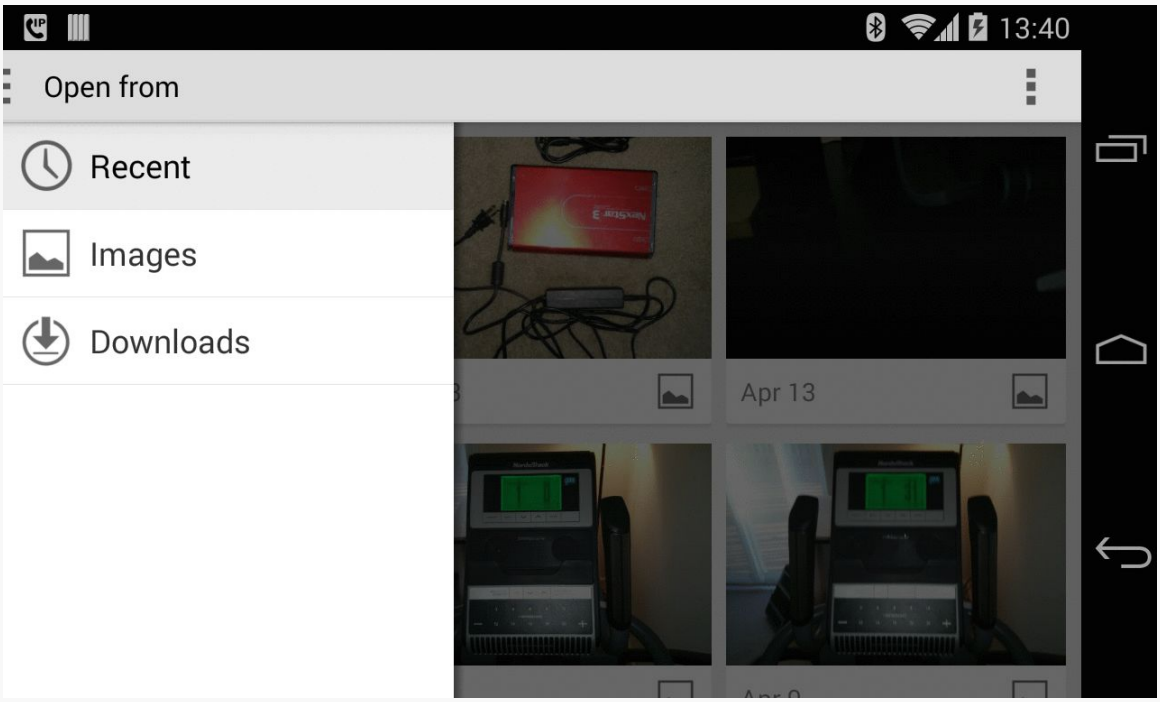


Figure 597: Storage Access Framework Picker, Showing Images

When the user taps on an image, the results wind up in our transcript UI:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

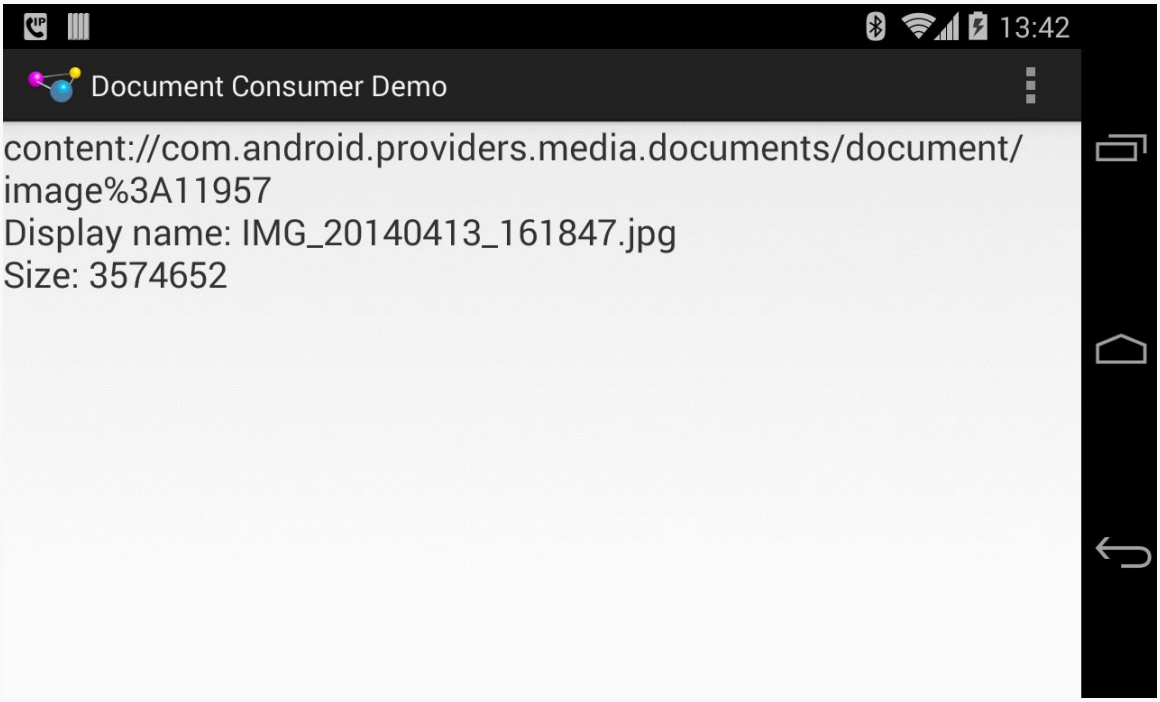


Figure 598: Uri, Display Name, and Size of Chosen File

Note that the default behavior of `ACTION_OPEN_DOCUMENT` is to let the user choose a single file. If your `Intent` includes `EXTRA_ALLOW_MULTIPLE`, set to `true`, then the user can choose multiple documents. Rather than getting their `Uri` values via `getData()` on the result `Intent`, you will need to call `getClipData()` on the `Intent` and iterate over the "clipboard entries".

The `Uri` itself can then be used to get an `InputStream` or `OutputStream` for the contents, using `openInputStream()` and `openOutputStream()` on `ContentResolver`, respectively. Note, though, that you cannot pass the `Uri` to other applications, as they may not have rights to work with that document the way that you do.

The Rest of the CRUD

`ACTION_OPEN_DOCUMENT` will give you a `Uri` for a document that you can open for reading — the "R" in "CRUD".

However, the other CRUD operations are also entirely possible.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Create

ACTION_CREATE_DOCUMENT will give you a Uri for a document that you can open for writing, as it is your document.

To do this, construct an Intent with:

- an action of ACTION_CREATE_DOCUMENT
- a category of CATEGORY_OPENABLE
- the MIME type of the content you wish to write
- an extra, named EXTRA_TITLE, containing your desired filename

Then, invoke startActivityForResult() on that Intent, and use the Uri supplied in the result Intent delivered to onActivityResult().

Update

The Uri returned from an ACTION_OPEN_DOCUMENT request may be writable. If it is, you can use openOutputStream() on a ContentResolver to write to that document. You can determine if a document is writable by examining the COLUMN_FLAGS value returned from a query() on the Uri — if it includes FLAG_SUPPORTS_WRITE, you can write to the document.

Delete

Similarly, if the COLUMN_FLAGS value includes FLAG_SUPPORTS_DELETE, you can delete the document by calling the static deleteDocument() method on the DocumentsContract class, supplying a ContentResolver plus the Uri of the document to be deleted.

Pondering Persistent Permissions

By default, you will have the rights to read (and optionally write) to the document represented by the Uri until the device is rebooted. That may be sufficient for your needs.

If, however, you need the rights to survive a reboot, you can call takePersistableUriPermission() on a ContentResolver, indicating the Uri of the document and the permissions (FLAG_GRANT_READ_URI_PERMISSION and/or

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

FLAG_GRANT_WRITE_URI_PERMISSION) that you want persisted. Those rights will then survive a reboot. However:

- They will not survive the document being deleted, so just because you have a saved Uri, do not assume that the Uri will still be valid
- You can revoke those rights by calling releasePersistableUriPermission() later on

In addition, you can call getPersistedUriPermissions() to find out what persisted permissions your app has. This returns a List of UriPermission objects, where each one of those represents a Uri, what persisted permissions (read or write) you have, and when the permissions were persisted.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Providing Documents

The Storage Access Framework gives developers access to `ACTION_OPEN_DOCUMENT` and related `Intent` actions to perform operations on a document provider.

However, what if you want to *be* a document provider?

To do that, you will need to create a subclass of `DocumentsProvider`, override some abstract methods, and perhaps put up with some really obtuse error messages.

This chapter will help you in setting up your `DocumentsProvider`. With luck, you will escape without encountering errors.

Prerequisites

This chapter assumes that you have read the preceding chapter on consuming documents, along with its prerequisites.

Have Your Content, and Provide it Too

Most apps will not need to implement a document provider. They might not even consume documents, let alone provide them to other apps.

However, if your app has document-style content, and that content is of a MIME type that could reasonably be manipulated by other apps, you should consider implementing a document provider to allow the user to manipulate that content using those other apps.

Historically, developers had two main approaches for content:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. Store it on internal storage
2. Store it on external storage

The internal storage route would be for content that would not normally be user-accessible, while external storage would be for user-accessible content.

In either of those cases, you will want to consider creating a document provider. In the case of internal storage, the user has no way to get to that content except through your app, and so if your app does not offer some capabilities that other apps do for that content, you limit your user by not having a document provider. In the case of external storage, while users in theory could use a file manager or something to try to get other apps to recognize the content, it will be easier for users if you provide a document provider to proactively publish this content to consuming apps.

However, bear in mind that your app does not have to store its content in either of these places. It could store the content in *somebody else's* document provider, using the mechanisms discussed in the preceding chapter on consuming documents. In this case, you would not need to publish a document provider yourself, as the content is available through the same provider that your code is using.

It may also be the case that while your app is the one directly storing the content on internal or external storage, that content would not reasonably be used by other apps. Perhaps it is in some non-standard format that other apps are unlikely to support. Perhaps the files are not to be used by other apps for security reasons. In these cases, skipping the document provider is reasonable, though not exactly ideal from the user's standpoint. In particular, it restricts them from using those files with apps that can work with any file, such as attaching them to email messages.

So, for example:

- If you are implementing a camera app, and you are not storing the photos in the standard `DIRECTORY_DCIM` location for photos, but you are storing the photos yourself in files consider implementing a document provider so users can get at the photos you are taking. But, if you are implementing a camera app, you might elect to allow the user to indicate some place in somebody else's document provider where you could save the photo on their behalf.
- If you are implementing some sort of network-synced file service (e.g., DropBox, Bittorrent Sync), or some sort of on-device version control system, consider implementing a document provider so users can manipulate the documents that you are managing on their behalf.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Key Provider Concepts

Creating a document provider is significantly more complex than is creating a document consumer. To help make sense of what is required, here are some key terms that you will need to understand.

Roots

A document provider can publish one or more “roots”. Basically, a “root” points to a tree of documents. Many providers will have just one root, but it is entirely possible for your provider to have more than one.

Documents

Documents, in turn, represent what in filesystem terms would be considered files and directories. A document can either have children (e.g., a directory) or it can have content (e.g., a file), but not both.

A root *also* is a document — here, “root” refers to the root of a tree of documents.

Root and Document IDs

Each root has an ID unique to your app to identify that root as being distinct from any other root. This is a string.

Each document — files and directories alike — will have a string document ID to uniquely identify that document within its root. If you have some natural identifier (e.g., a primary key in some table), feel free to use it. Otherwise, you might consider your document ID to be some path to get to the document.

And, since a root is also a document, a root will have both a root ID and a document ID.

These document IDs need to be durable. Clients, or the Storage Access Framework itself, may wind up caching these IDs. Hence, pick something that not only uniquely identifies this document, but will *continue* to uniquely identify the document even after the document has been modified.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Pieces of a Provider

The Documents/Provider sample application implements a document provider, one that serves documents baked into the app itself in `assets/`. Of course, this is a rather artificial scenario — usually, a document provider will be working with a read/write data store, like internal storage.

The Activity

A document provider app probably needs a real UI. Perhaps that UI is for broader functionality that the app provides on top of serving documents. Perhaps that UI is merely to configure the document store, such as providing account credentials for the online storage service that the document provider exposes on Android.

If nothing else, you will need a do-nothing activity for the user to run, to ensure that your app is moved out of the stopped state.

In the sample app, this is handled by `MainActivity`, which uses `Theme.NoDisplay` to eschew a UI and just shows a `Toast` to indicate that the provider is now activated:

```
package com.commonsware.android.documents.provider;

import android.app.Activity;
import android.content.res.AssetManager;
import android.os.Bundle;
import android.util.Log;
import android.widget.Toast;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;

public class MainActivity extends Activity {
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Toast.makeText(this, R.string.activated, Toast.LENGTH_LONG).show();
    finish();
  }
}
```

The API Level Resources

The document provider itself comes courtesy of a subclass of `DocumentsProvider`. However, the `DocumentsProvider` class only exists on API Level 19 and higher — our

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

subclass is useless on older devices. To ensure that our provider is only used on API Level 19 and higher, we should only enable it on API Level 19+ devices.

To that end, in `res/values/bools.xml`, we have a boolean resource named `min19`, set to `false`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool name="min19">false</bool>
</resources>
```

In `res/values-v19/bools.xml`, we redefine that boolean resource to be `true`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool name="min19">true</bool>
</resources>
```

Hence, when we refer to the `min19` boolean resource, we will get `true` or `false` depending upon whether we are on API Level 19 or not.

The Manifest

Since `DocumentsProvider` is a subclass of `ContentProvider`, we will need a `<provider>` element in the manifest pointing to our subclass of `DocumentsProvider`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.documents.provider"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="19"
    android:targetSdkVersion="19"/>

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name="MainActivity"
      android:label="@string/app_name"
            android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      </intent-filter>
    </activity>
        <provider
            android:name=".DemoDocumentsProvider"
            android:authorities="com.commonsware.android.documents.provider"
            android:grantUriPermissions="true"
            android:exported="true"
            android:permission="android.permission.MANAGE_DOCUMENTS"
            android:enabled="@bool/min19">
            <intent-filter>
                <action android:name="android.content.action.DOCUMENTS_PROVIDER"
/>
            </intent-filter>
        </provider>
    </application>

</manifest>
```

That element:

- Uses our `@bool/min19` resource from above to indicate that this component should only be enabled on API Level 19 and higher
- Is exported, but requires that applications looking to talk to our provider hold the `MANAGE_DOCUMENTS` permission, which can only be held by the firmware (or apps signed with the firmware's signing key)
- Sets the `android:grantUriPermissions` attribute to `true`, as that will be used by `DocumentsProvider` to allow third-party apps limited, conditional access to our documents
- Has your standard `android:name` and `android:authorities` attributes, as with any other `<provider>`

In addition, the `<provider>` has a nested `<intent-filter>` element. This may seem odd, as this used to be impossible, and it is not intuitively obvious what it would mean for a `ContentProvider` to have an `IntentFilter`. It also is not documented as being allowed on `<provider>`, so we have no official explanation of what this means. Most likely, the magic `android.content.action.DOCUMENTS_PROVIDER` filter is being used simply as a marker, to indicate to Android that this particular `<provider>` is part of the Storage Access Framework and implements a `DocumentsProvider`.

The DocumentsProvider

The real business logic of publishing documents comes from your subclass of `DocumentsProvider`. As this class is new to API Level 19, your build target (e.g., `compileSdkVersion` in `build.gradle`) needs to be 19 or higher.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A minimal `DocumentsProvider` implementation will typically need five methods, outlined below.

onCreate()

As with any `ContentProvider`, your `DocumentsProvider` can override `onCreate()` to perform initialization work. Technically, this is not required, but the odds are very good that you will have *something* that you need to initialize.

In the case of our sample `DocumentsProvider` — named `DemoDocumentsProvider` — `onCreate()` simply obtains access to an `AssetManager` instance that can be used for serving documents:

```
private AssetManager assets;

@Override
public boolean onCreate() {
  assets=getContext().getAssets();

  return(true);
}
```

queryRoots()

Your `queryRoots()` method needs to return information about the root(s) that your provider will provide.

However, rather than returning this in the form of some clean object model (e.g., a `List` of `Document.Root` objects or some such), the return value is a `Cursor`. While in principle this `Cursor` could come from a database, in many cases it will be a `MatrixCursor`, which is a `Cursor` interface over a two-dimensional array representing the rows and columns.

From here, you should return all *presently valid* roots. The "presently valid" part is because a root might exist but not be usable at the present time. For example, suppose that you are writing a `DocumentsProvider` that provides a document interface to an Internet-hosted storage service. In this case, you may need the user to authenticate in order to allow access to those files, such as to pass that authentication data along to the Web service to be able to retrieve directory and file data. If the user is not presently logged in, though, not only can you not talk to the Web service right now, but you do not have the ability to force the user to authenticate right now. Instead, you will have to cull the root(s) governed by those

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

authentication credentials. This may mean that the `Cursor` you return has no rows, as you simply do not have anything that can be published right now.

The `Cursor` that you return will have one row per presently valid root. The columns will be ones defined on the `DocumentsContract.Root` class. Your `queryRoots()` method is passed a `String` array representing the columns requested by the Storage Access Framework. As your app may not support all of those columns, you will need to determine the intersection between the requested columns and the ones you support.

The sample app defines a `SUPPORTED_ROOT_PROJECTION` static data member to list the `DocumentsContract.Root` columns that are supported in general:

```
  private static final String[] SUPPORTED_ROOT_PROJECTION=new String[] {
      Root.COLUMN_ROOT_ID, Root.COLUMN_FLAGS, Root.COLUMN_TITLE,
      Root.COLUMN_DOCUMENT_ID, Root.COLUMN_ICON };
```

And the demo provider has a private `netProjection()` utility method that computes the intersection between the requested columns and the supported ones:

```
private static String[] netProjection(String[] requested, String[] supported) {
  if (requested==null) {
    return(supported);
  }

  ArrayList<String> result=new ArrayList<String>();

  for (String request : requested) {
    for (String support : supported) {
      if (request.equals(support)) {
        result.add(request);
        break;
      }
    }
  }

  return(result.toArray(new String[0]));
}
```

That net projection is used in the `MatrixCursor` constructor, to teach it the available columns, as part of the `queryRoots()` implementation:

```
  @Override
  public Cursor queryRoots(String[] projection)
      throws FileNotFoundException {
    String[] netProjection=
        netProjection(projection, SUPPORTED_ROOT_PROJECTION);
    MatrixCursor result=new MatrixCursor(netProjection);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    MatrixCursor.RowBuilder row=result.newRow();

    row.add(Root.COLUMN_ROOT_ID, ROOT_ID);
    row.add(Root.COLUMN_ICON, R.drawable.ic_launcher);
    row.add(Root.COLUMN_FLAGS, Root.FLAG_LOCAL_ONLY);
    row.add(Root.COLUMN_TITLE, getContext().getString(R.string.root));
    row.add(Root.COLUMN_DOCUMENT_ID, ROOT_DOCUMENT_ID);

    return(result);
  }
```

`queryRoots()` then adds a row to the `MatrixCursor`, through a `MatrixCursor.RowBuilder`, containing five columns:

1. `DocumentsContract.Root.COLUMN_ROOT_ID` is the root ID for this root, as described earlier in this chapter.
2. `DocumentsContract.Root.COLUMN_ICON`, which is a reference to a drawable resource that may be used in Storage Access Framework UI to help visually represent this root. In principle, this could be anything; in practice, you will probably choose your launcher icon, as it is the icon that the user will recognize.
3. `DocumentsContract.Root.COLUMN_FLAGS`, indicating which optional capabilities this root supports. In this case, the only flag we are setting is `FLAG_LOCAL_ONLY`, indicating that network I/O is not required to browse the contents of the provider. Our sample app indicates that it is local-only, as its documents are all packaged in `assets/`. A provider backed by a Web service, though, would not include this flag, so the Storage Access Framework knows that calls to some of the other methods (e.g., `queryChildDocuments()`) may take a significant amount of time.
4. `DocumentsContract.Root.COLUMN_TITLE`, which is a string identifying this root. The title and icon will tend to be included in Storage Access Framework-supplied UIs. In this case, with only a single root, the title is hard-coded to be a string resource. In other cases, this might be some other human-grokkable display name (e.g., the name of some storage service account).
5. `DocumentsContract.Root.COLUMN_DOCUMENT_ID`, which returns the document ID representing the document tree for this root.

In this case, the document IDs for this `DocumentsProvider` are the relative paths within `assets/` of the files, starting from a root `docs/` directory. So, while the root ID could be anything, the root document ID should be consistent with the other document ID values. In this case, the sample app uses:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private static final String ROOT_ID="thisIsMyBoomstick";
  private static final String ROOT_DOCUMENT_ID="docs";
```

queryChildDocuments()

As noted previously, some documents will represent a directory, while others will represent files. For those that represent a directory, `queryChildDocuments()` will need to return the document information for the contents of the directory.

`queryChildDocuments()` is passed:

- the document ID of the directory
- the columns, defined on the `DocumentsContract.Document` class, that the Storage Access Framework wants
- the sort order, expressed as a SQL-style `ORDER BY` clause (minus the actual `ORDER BY` part), that you might use to help control the order in which to return the child documents (or ignore if you wish)

As with `queryRoots()`, we need to come up with the intersection of the columns that the requester asks for and the columns that we support. There is a `static` string array named `SUPPORTED_DOCUMENT_PROJECTION` that represents the columns that we support:

```
  private static final String[] SUPPORTED_DOCUMENT_PROJECTION=
      new String[] { Document.COLUMN_DOCUMENT_ID, Document.COLUMN_SIZE,
          Document.COLUMN_MIME_TYPE, Document.COLUMN_DISPLAY_NAME,
          Document.COLUMN_FLAGS};
```

The `queryChildDocuments()` method then uses the same `netProjection()` helper method that `queryRoots()` did to determine the intersection:

```
  @Override
  public Cursor queryChildDocuments(String parentDocId,
                                    String[] projection,
                                    String sortOrder)
      throws FileNotFoundException {
    String[] netProjection=
        netProjection(projection, SUPPORTED_DOCUMENT_PROJECTION);
    MatrixCursor result=new MatrixCursor(netProjection);

    try {
      String[] children=assets.list(parentDocId);

      for (String child : children) {
        addDocumentRow(result, child,
                       parentDocId + File.separator + child);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      }
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(),
            "Exception reading asset dir", e);
    }

    return(result);
  }
```

As with `queryRoots()`, the return value of `queryChildDocuments()` is a `Cursor` representing the documents contained in this directory. Once again, we use a `MatrixCursor` to build up an in-memory `Cursor`, this time for all files within the `assets/` directory denoted by `parentDocId`, using the `list()` method on `AssetManager` to find out what those files are.

The logic to populate the `MatrixCursor` is delegated to an `addDocumentRow()` private method, as we will be using it elsewhere in this `DocumentsProvider` implementation. `addDocumentRow()` creates a `MatrixCursor.RowBuilder` and fills in the supported columns:

```
  private void addDocumentRow(MatrixCursor result, String child,
                              String assetPath) throws IOException {
    MatrixCursor.RowBuilder row=result.newRow();

    row.add(Document.COLUMN_DOCUMENT_ID, assetPath);

    if (isDirectory(assetPath)) {
      row.add(Document.COLUMN_MIME_TYPE, Document.MIME_TYPE_DIR);
    }
    else {
      String ext=MimeTypeMap.getFileExtensionFromUrl(assetPath);

      row.add(Document.COLUMN_MIME_TYPE,
          MimeTypeMap.getSingleton().getMimeTypeFromExtension(ext));
      row.add(Document.COLUMN_SIZE, getAssetLength(assetPath));
    }

    row.add(Document.COLUMN_DISPLAY_NAME, child);
    row.add(Document.COLUMN_FLAGS, 0);
  }
```

Of note:

- the document ID of the child is simply its relative path within `assets/`
- the MIME type is a special one if the child document represents a directory, or else is looked up using `MimeTypeMap` if the child document represents a file

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- the "display name" could be something special (e.g., the `<title>` of a Web page), but in this case is just the filename

To determine if an asset path represents a directory, the `isDirectory()` utility method just sees if `list()` returns a non-empty list:

```
  private boolean isDirectory(String assetPath) throws IOException {
    return(assets.list(assetPath).length>=1);
  }
```

To find the size of a document — to fill in the `COLUMN_SIZE` column in the output — we can ask the `AssetManager` for a `FileDescriptor` on the asset, then obtain the length from that descriptor, as seen in the `getAssetLength()` utility method:

```
  private long getAssetLength(String assetPath) throws IOException {
    return(assets.openFd(assetPath).getLength());
  }
```

The net result is that, given the name of a directory in `assets/`, we return a `Cursor` with one row per child of that directory, with columns indicating details of that child.

queryDocument()

`queryDocument()` is similar to `queryChildDocuments()`. Both return a `Cursor` with the same sorts of columns as output. The difference: `queryDocument()` provides you with the document ID of a file, and you return details of that file. By contrast, `queryChildDocuments()` gives you the document ID of a directory, and you return the details of all documents within that directory.

This is why `addDocumentRow()` was implemented as a separate method, as we need the same business logic (populate a `MatrixCursor` row based on an asset path) from `queryDocument()`:

```
  @Override
  public Cursor queryDocument(String documentId, String[] projection)
      throws FileNotFoundException {
    String[] netProjection=
        netProjection(projection, SUPPORTED_DOCUMENT_PROJECTION);
    MatrixCursor result=new MatrixCursor(netProjection);

    try {
      addDocumentRow(result, Uri.parse(documentId).getLastPathSegment(),
          documentId);
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception reading asset dir", e);
    }

    return(result);
  }
```

In this case, the only thing different is that we need to get the bare filename, for use in the `DISPLAY_NAME` field. Here, we cheat a bit and use `getLastPathSegment()` on `Uri` to obtain the filename.

openDocument()

The `openDocument()` method behaves much like the `openFile()` method of a classic streaming `ContentProvider`: given a path, you return a `ParcelFileDescriptor` representing the file contents. For documents that are true files on the filesystem, you can use the static `open()` method on `ParcelFileDescriptor`. For documents that are not files on the filesystem — such as documents that are assets in the APK — you will need to set up a `ParcelFileDescriptor` pipe and stream the content that way.

That is what `DemoDocumentsProvider` does, using logic copied from the book's streaming `ContentProvider` samples:

```
  @Override
  public ParcelFileDescriptor openDocument(String documentId,
                                           String mode,
                                           CancellationSignal signal)
      throws FileNotFoundException {
    ParcelFileDescriptor[] pipe=null;

    try {
      pipe=ParcelFileDescriptor.createPipe();
      AssetManager assets=getContext().getResources().getAssets();

      new TransferThread(assets.open(documentId),
          new ParcelFileDescriptor.AutoCloseOutputStream(pipe[1])).start();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception opening pipe", e);
      throw new FileNotFoundException("Could not open pipe for: "
          + documentId);
    }

    return(pipe[0]);
  }
```

`openDocument()` is passed three parameters:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. The document ID of the document to stream back
2. A file mode (`r`, `w`, or `wt`) indicating what sort of operations the client wants to perform on the stream
3. A `CancellationSignal` that we can use to find out that our streaming is being interrupted

In this case:

- `openDocument()` ignores the `mode`, because it did not return `FLAG_SUPPORTS_WRITE` in either `queryChildDocuments()` or `queryDocument()` to indicate that writing is an option, so the `mode` should always be `r`
- `openDocument()` ignores the `CancellationSignal`, though in reality it should pay attention to it when streaming back the content and stop streaming when requested

The `TransferThread` that does the actual streaming is, once again, the same as the one used earlier in this book for a streaming `ContentProvider`:

```
  static class TransferThread extends Thread {
    InputStream in;
    OutputStream out;

    TransferThread(InputStream in, OutputStream out) {
      this.in=in;
      this.out=out;
    }

    @Override
    public void run() {
      byte[] buf=new byte[8192];
      int len;

      try {
        while ((len=in.read(buf)) >= 0) {
          out.write(buf, 0, len);
        }

        in.close();
        out.flush();
        out.close();
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(),
            "Exception transferring file", e);
      }
    }
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Results

If you have both this sample app and the one from the previous chapter, then run the one from the previous chapter to bring up the Storage Access Framework UI, you will see our provider among the list of available providers:

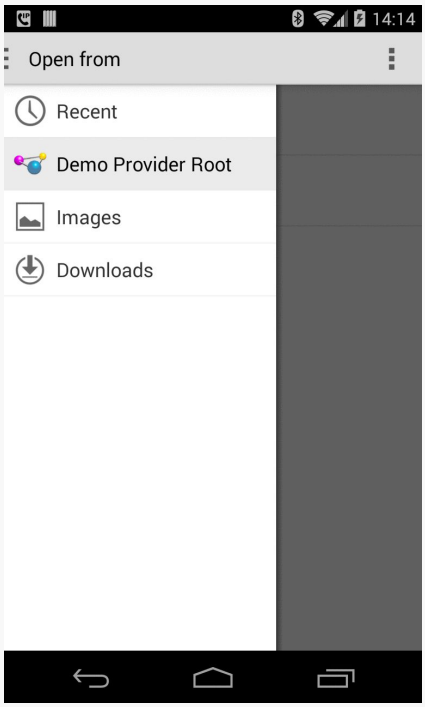


Figure 599: Storage Access Framework Picker, Showing Custom Provider

The provider's `assets/docs/` directory contains three files, one just off the root and two in a `bar/` subdirectory:

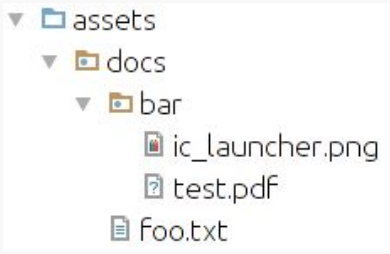


Figure 600: DocumentsProvider Sample Documents

Hence, tapping on our provider in the Storage Access Framework picker brings up the contents of the root document:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

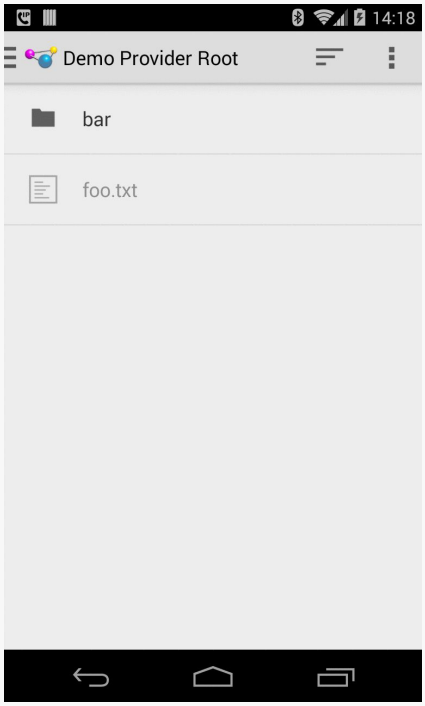


Figure 601: Storage Access Framework Picker, Showing Documents in Root

Tapping on the `bar/` directory brings up its contents in turn:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

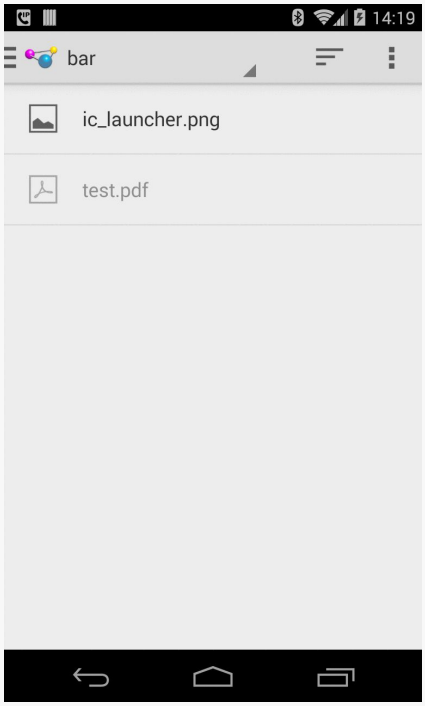


Figure 602: Storage Access Framework Picker, Showing Yet More Documents

Tapping on one of the files brings up the details for that file:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

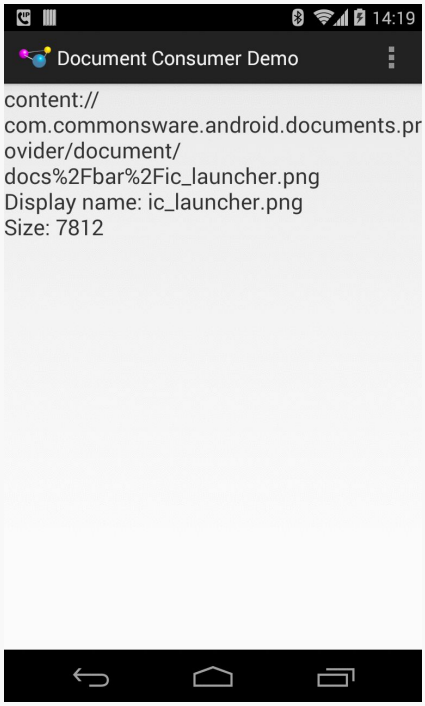


Figure 603: Document Consumer, Showing Details of Picked Document

Optional Provider Capabilities

A `DocumentsProvider` can do a fair bit more than what the above sample app demonstrates. While the sample will suffice for the basics, it is reasonably likely that a production-grade `DocumentsProvider` will need to implement and provide some other optional capabilities, such as those described in this section.

Other CRUD Operations

CRUD — Create, Read, Update, and Delete — is a standard shorthand for the basic operations one can perform on data. The sample app handles the "Read" portion of CRUD, but a `DocumentsProvider` can support all of them if desired.

Create

It may be that your `DocumentsProvider` is only going to serve up documents that were created in your app, or were created outside of the Android device (e.g., on a Web app). If, however, you want consumers of your provider to be able to use

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

ACTION_CREATE_DOCUMENT to create new documents in your provider, you will need to do a few things.

First, in the COLUMN_FLAGS for the relevant root(s) returned by queryRoots(), you will need to include FLAG_SUPPORTS_CREATE, defined on DocumentsContract.Root. This indicates that at least one directory within that root supports creating new documents. Without this flag, your root(s) will be shown for ACTION_OPEN_DOCUMENT requests but not ACTION_CREATE_DOCUMENT requests.

Next, in one or more directories returned as part of queryDocument() and queryChildDocuments() calls, in the COLUMN_FLAGS column, you will need to include FLAG_DIR_SUPPORTS_CREATE, defined on DocumentsContract.Documents. This indicates that this document is a directory that supports creating new documents inside of it. Otherwise, a directory will be assumed to not support creating new documents. Note that this flag is only used for documents representing directories, not documents representing files.

Finally, you will need to implement createDocument() in your DocumentsProvider. This will be called if a consumer app used ACTION_CREATE_DOCUMENT and the user chose your provider and one of your directories for the new document. You are passed in:

- the document ID of the directory
- the MIME type of the new file
- a suggested display name to use for the new file, though you can modify this if needed

Your job, in createDocument(), is to create the document and return the document ID for the newly-created document. For example, if your documents are held in internal storage, you might create a new file for the document itself plus a database row in some documents table to hold the MIME type and display name.

Update

For something like files, an "update" is replacing the current contents with something new. In the case of a streaming protocol like DocumentsProvider, this implies that your provider can support output as well as input.

This too requires a few changes to your provider.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

First, for the file(s) that can be updated, in the results for `queryDocument()` and `queryChildDocuments()`, you will need to include `FLAG_SUPPORTS_WRITE` in `COLUMN_FLAGS`, to indicate that writing to this file should work.

Then, you will need to pay attention to the `mode` passed into `openDocument()`. If the mode is `w` or `wt`, you would need to arrange to support writing the file, where your background thread reads from an `InputStream` on the pipe and writes the data to wherever your data is being stored.

Delete

For any documents that the consumer can delete, include `FLAG_SUPPORTS_DELETE` in `COLUMN_FLAGS` in the results for `queryDocument()` and `queryChildDocuments()`.

You will also need to implement `deleteDocument()` in your `DocumentsProvider`. You are supplied the document ID to delete, and your job is to delete it.

If the document represents a directory, you may also need to delete all of its children. That really depends on how you are leveraging the "directory" construct in `DocumentsProvider`:

- If the "directory" is like a filesystem directory, where children have only one parent, you will want to delete the children when you delete the parent
- If the "directory" is more like a category, such as a tag, where children could have multiple parents, you will need to decide how to handle the children that would be orphaned by your deleting the last parent (delete the children? move them to some other default parent? something else?)

Change Notification

If the data served by your provider changes, it is incumbent upon you to let possible consumers know about the change. For example, if you elect to delete children when you delete their parent, you should let consumers know that those children were deleted. This is not necessary for direct operations performed by consumers (e.g., writing to a document), but is necessary for anything else.

To do that, you call `notifyChange()` on a `ContentResolver`, just as you would for changes to the data in a `ContentProvider`. However, `notifyChange()` takes a `Uri` as a parameter, to indicate the scope of the change. There are `static` utility methods on `DocumentsContract` that will return a `Uri` that you can use. Notably:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `buildDocumentUri()`, given your authority and a document ID, provides a `Uri` that points to that document
- `buildChildDocumentsUri()`, given your authority and a document ID, provides a `Uri` that represents the collection of children of that document

So, for example, if by deleting a parent you also delete the children, you would use `buildChildDocumentsUri()` with `notifyChange()` to ensure that consumers know that those children were modified. The Storage Access Framework will use methods like `queryChildDocuments()` to determine that the children were deleted in this case.

Thumbnails

By default, the Storage Access Framework will use stock icons for directories and files. You can supply your own thumbnails instead, though, if you want. To do this:

- Include `FLAG_SUPPORTS_THUMBNAIL` in the `COLUMN_FLAGS` for the affected document(s) in `queryDocument()` and `queryChildDocuments()`
- Implement `openDocumentThumbnail()` in your `DocumentsProvider`

`openDocumentThumbnail()` is provided the document ID of the document whose thumbnail is required, along with a `Point` object providing a requested size. While your thumbnail does not have to exactly match that size — for example, the aspect ratio that is requested may not match the thumbnail — it should be close.

However, the return value for `openDocumentThumbnail()` is an `AssetFileDescriptor`, which is a wrapper around a `ParcelFileDescriptor`. If your image exists as a file that happens to be the right size, return it by using the static `open()` method on `ParcelFileDescriptor` is fairly straightforward. If, however, you need to scale your source image to fit the desired size, implementing this via a pipe will be moderately tedious.

Recent Documents

If your app has its own concept of recent documents, you can expose that roster to the Storage Access Framework, which can incorporate it as part of its UI. To do this:

- Have your `queryRoots()` method include `FLAG_SUPPORTS_RECENTS` in the `COLUMN_FLAGS` value for the root(s) that support recent documents
- Implement `queryRecentDocuments()` on your `DocumentsProvider`, where you are given the root ID (*not* a document ID!) of one of your roots, and you

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

need to return the same sort of `Cursor` as you would from `queryChildDocuments()`, but representing the recent documents for that root

The two constraints upon the returned `Cursor` are:

- It should be sorted descending based on last-modified date (e.g., the `COLUMN_LAST_MODIFIED` column in your rows)
- It should be capped at 64 rows, though it can be less if desired

Note that you can, if you wish, have the `Cursor` return rows reflecting both files and directories — you are not limited to one or the other.

Search

If your provider has its own search capability, you can expose that to the Storage Access Framework, which in turn can make it available to users looking for a certain document. To support this:

- Have your `queryRoots()` method include `FLAG_SUPPORTS_SEARCH` in the `COLUMN_FLAGS` value for the root(s) that support searching
- Implement `querySearchDocuments()` on your `DocumentsProvider`, where you are given the root ID (*not* a document ID!) of one of your roots, and you need to return the same sort of `Cursor` as you would from `queryChildDocuments()`, but representing the results of a search

`querySearchDocuments()` is passed a `String` representing the search expression entered by the user. It is up to you to decide what that expression means. It is also up to you to determine where you are searching for that expression (filenames? file contents?).

Note that the `Cursor` you return should only contain documents that reflect files, not documents that point to directories.

Other Flags

There are a few other flags that are available to you on `DocumentsContract.Document` that you can use in `COLUMN_FLAGS` for `Cursor` results representing a document or collection of documents, such as the results of `queryDocument()` and `queryChildDocuments()`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- If the document represents a directory, and you are supporting thumbnails, and you would like the contents of this directory to be represented in a thumbnail grid as opposed to a list, include `FLAG_DIR_PREFERS_GRID`
- If the document represents a directory, and you feel that users will be better served showing the documents in descending order based upon `COLUMN_LAST_MODIFIED`, rather than alphabetical by display name, include `FLAG_DIR_PREFERS_LAST_MODIFIED`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Encrypted Storage

SQLite databases, by default, are stored on internal storage, accessible only to the app that creates them.

At least, that is the theory.

In practice, it is conceivable that others could get at an app's SQLite database, and that those "others" may not have the user's best interests at heart. Hence, if you are storing data in SQLite that should remain confidential despite extreme measures to steal the data, you may wish to consider encrypting the database.

Perhaps the simplest way to encrypt a SQLite database is to use SQLCipher. SQLCipher is a SQLite extension that encrypts and decrypts database pages as they are written and read. However, SQLite extensions need to be compiled into SQLite, and the stock Android SQLite does not have the SQLCipher extension.

SQLCipher for Android, therefore, comes in the form of a replacement implementation of SQLite that you add as an NDK library to your project. It also ships with replacement editions of the android.database.sqlite.* classes that use the SQLCipher library instead of the built-in SQLite. This way, your app can be largely oblivious to the actual database implementation, particularly if it is hidden behind a `ContentProvider` or similar abstraction layer.

SQLCipher for Android is a joint initiative of Zetetic (the creators of SQLCipher) and the Guardian Project (home of many privacy-enhancing projects for Android). SQLCipher for Android is open source, under the Apache License 2.0.

Many developers of enterprise-grade apps for Android (e.g., Salesforce.com, JPMorgan Chase) use SQLCipher for securing their apps.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- database access
- content provider theory
- content provider implementations

Scenarios for Encryption

So, why might you want to encrypt a database?

Some developers probably are thinking that this is a way of protecting the app's content against "those pesky rooted device users". In practice, this is unlikely to help. As with most encryption mechanisms, SQLCipher uses an encryption key. If the app has the key, such as being hard-coded into the app itself, anyone can get the key by reverse-engineering the app.

Rather, encrypted databases are to help the user defend their data against other people seeing it when they should not. The classic example is somebody leaving their phone in the back of a taxi — if that device winds up in the hands of some group with the skills to root the device, they can get at any unencrypted content they want. While some users will handle this via the whole-disk encryption available since Android 3.0, others might not.

If the database is going anywhere other than internal storage, there is all the more reason to consider encrypting it, as then it may not even require a rooted device to access the database. Scenarios here include:

1. Databases stored on external storage
2. Databases backed up using external storage, BackupManager, or another Internet-based solution
3. Databases explicitly being shared among a user's devices, or between a user's device and a desktop (note that SQLCipher works on many operating systems, including desktops and iOS)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Obtaining SQLCipher

SQLCipher for Android is available from its GitHub repository. The downloads area contains ZIP archives of what you need. As of September 2014, the current shipping version was 3.1.0.

NOTE: If you have been using SQLCipher prior to version 2.2.1, please upgrade to 3.0.x! Some modifications were made in July 2013 "to address a compatibility issue with an upcoming Android OS release" (presumably Android 4.4). You can read more in the SQLCipher project's blog post about the upgrade.

NOTE #2: If you have been using SQLCipher prior to version 3.0.x, and you have deployed your app in production, you will have some additional work to do to upgrade your database to the new file format. This is covered later in this chapter.

Attaching SQLCipher To Your Project

The way you get SQLCipher for Android set up for your project depends upon your IDE.

Android Studio

Someday, hopefully we will be able to get SQLCipher for Android via an artifact in a repository.

In the short term, though, you have to go through a manual setup, based in part on one of the NDK recipes mentioned in the chapter on the NDK:

- Copy the contents of the `assets/` directory from the ZIP file into your project's `assets/` directory in the `main/` sourceset (e.g., `app/src/main/assets/` from a typical project root directory)
- Copy the JARs from the ZIP file's `libs/` directories into a top-level `libs/` directory, as a peer to `app/` or other modules in the project
- Copy the `armeabi/`, `armeabi-v7a/`, and `x86/` directories from the ZIP file's `libs/` directories into a `jniLibs/` directory in your `main/` sourceset
- In your `build.gradle`, have the following line in your top-level `dependencies` closure:

```
compile fileTree(dir: 'libs', include: '*.jar')
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That tells Gradle for Android to pull in your JARs, and the `jniLibs/` directory tells Gradle for Android to package those pre-build NDK libraries with your app as well.

The `Database/ConstantsSecure-AndroidStudio` and `Database/SQLCipherPassphrase-AndroidStudio` sample projects demonstrate this setup.

Eclipse

Given an existing Android project, to use SQLCipher for Android, you need to extract the contents of the ZIP archive's `libs/` directory and put them in your own project's `libs/` directory (creating the latter if needed). The ZIP archive's `libs/` directory contains a few JARs, plus a set of NDK-compiled C/C++ libraries for SQLite with the SQLCipher extension.

You will also need to copy the contents of the ZIP archive's `assets/` folder into your project's `assets/` folder (creating the latter if needed).

Using SQLCipher

If you have existing code that uses classic Android SQLite, you will need to change your import statements to pick up the SQLCipher for Android equivalents of the classes. For example, you obtain `SQLiteDatabase` now from `net.sqlcipher.database.sqlcipher`, not `android.database.sqlite`. Similarly, you obtain `SQLException` from `net.sqlcipher.database` instead of `android.database`. Unfortunately, there is no complete list of which classes need this conversion — `Cursor`, for example, does not. Try converting everything from `android.database` and `android.database.sqlite`, and leave alone those that do not exist in the SQLCipher for Android equivalent packages.

Before starting to use SQLCipher for Android, you need to call `SQLiteDatabase.loadLibs()`, supplying a suitable `Context` object as a parameter. This initializes the necessary libraries. If you are using a `ContentProvider`, just call this in `onCreate()` before actually using anything else with your database. If you are not using a `ContentProvider`, you probably will want to create a custom subclass of `Application` and make this call from that class' `onCreate()`, and reference your custom `Application` class in the `android:name` attribute of the `<application>` element in your manifest. Either of these approaches will help ensure that the libraries are ready before you try doing anything with the database.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Finally, when calling `getReadableDatabase()` or `getWritableDatabase()` on `SQLiteDatabase`, you need to supply the encryption key to use. This may be somewhat tricky with a `ContentProvider`, as there is not an obvious way for you to get the key to the provider in advance of accessing the database. On older versions of Android, you will probably wind up using a static data member — before trying to first use the database, get the encryption key (e.g., based on a password typed by the user), put it in a static data member, and have your `ContentProvider` read the value from there.

However, starting with API Level 11, there is another approach for interacting with a `ContentProvider` beyond the scope of the traditional `query()`, `insert()`, etc. methods, by means of the `call()` method. That is the approach taken in the `Database/ConstantsSecure` sample app (and the `Database/ConstantsSecure-AndroidStudio` equivalent for Android Studio), yet another variation of the `ConstantsBrowser`, but where the information is stored in a SQLCipher for Android database, to keep our precious gravitational constants away from those who might abuse them.

Our revised `Provider` class switches its imports to the ones needed by SQLCipher for Android:

```
import net.sqlcipher.SQLException;
import net.sqlcipher.database.SQLiteDatabase;
import net.sqlcipher.database.SQLiteQueryBuilder;
```

In `onCreate()`, it initializes the libraries before creating our `DatabaseHelper` (our `SQLiteOpenHelper` implementation):

```
  @Override
  public boolean onCreate() {
    SQLiteDatabase.loadLibs(getContext());
    dbHelper=(new DatabaseHelper(getContext()));

    return((dbHelper == null) ? false : true);
  }
```

That `DatabaseHelper` is unchanged from previous editions, other than altering the imports to use SQLCipher for Android equivalents. However, in order to really use this `DatabaseHelper`, we will need the user's passphrase, which is set via the `call()` mechanism.

A client can call `call()` on a `ContentProvider` by means of a `ContentResolver`. `call()` takes the `Uri` of the `ContentProvider`, the name of the "method" to call, and

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

an optional `String` argument and `Bundle` for additional parameters. For example, our `ConstantsBrowser` activity now uses `call()` to call a "method" on the `Provider`, supplying the encryption key (here hard-coded for simplicity):

```
    getContentResolver().call(Provider.Constants.CONTENT_URI,
                              Provider.SET_KEY_METHOD, "sekrit", null);
```

That `call()` routes to a `call()` implementation on the `ContentProvider`, which is supplied all of the parameters except the `Uri`. It is up to the `ContentProvider` to examine the "method" name and handle it accordingly, optionally returning a `Bundle` of return values. In the case of our `Provider` class, if the call is for setting the encryption key, and the supplied key is not `null`, we use it to access our `SQLiteDatabase`:

```
  @Override
  public Bundle call(String method, String arg, Bundle extras) {
    if (SET_KEY_METHOD.equals(method) && arg != null) {
      db=dbHelper.getWritableDatabase(arg);
    }

    return(null);
  }
```

This way, we supply the passphrase to the `ContentProvider` before trying to use the database, while also avoiding a static data member. The downside of this approach, or pretty much anything involving SQLCipher for Android, is that we have to be very careful to ensure that we are routing the user through the login process before trying to use the database, no matter how they enter our app (launcher icon, recent tasks, app widget tap, started by a third-party app, etc.). While in this case, our hard-coded passphrase avoids this complexity, it results in extremely weak security, as anyone could decompile the app, find the passphrase, and decrypt the database. We will touch on keys and passwords more later in this chapter.

Note that we do not hold onto the passphrase, only the opened database. This is for added security, so somebody with a debugger cannot attach to our process and inspect the `ContentProvider` to find the passphrase. Of course, in this particular implementation, they could just as easily find where we hard-coded it.

The implementation of methods like `query()` then use the already-opened `SQLiteDatabase`:

```
  @Override
  public Cursor query(Uri url, String[] projection, String selection,
                      String[] selectionArgs, String sort) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    SQLiteQueryBuilder qb=new SQLiteQueryBuilder();

    qb.setTables(TABLE);

    String orderBy;

    if (TextUtils.isEmpty(sort)) {
      orderBy=Constants.DEFAULT_SORT_ORDER;
    }
    else {
      orderBy=sort;
    }

    Cursor c=
        qb.query(db, projection, selection, selectionArgs, null, null,
                 orderBy);

    c.setNotificationUri(getContext().getContentResolver(), url);

    return(c);
  }
```

Nothing else in the `ConstantsBrowser` activity needs to change, because the `ContentProvider` facade hides the rest of the implementation details.

SQLCipher Limitations

Alas, SQLCipher for Android is not perfect.

It will add a few MB to the size of your APK file per CPU architecture. For most modern Android devices, this extra size will not be a huge issue, though it will be an impediment for older devices with less internal storage, or for apps that are getting close to the size limits imposed by the Play Store or other distribution mechanisms. The chapter on the NDK contains a section about a technology called `libhoudini` that can help reduce this bloat, albeit with a significant performance penalty.

However, the size is mostly from code, and that may cause a problem for Eclipse users. Eclipse may crash with its own `OutOfMemoryError` during the final build process. To address that, find your `eclipse.ini` file (location varies by OS and installation method) and increase the `-Xmx` value shown on one of the lines (e.g., change it to `-Xmx512m`).

Other code that expects to be using native SQLite databases will require alteration to work with SQLCipher for Android databases. For example, the `SQLiteAssetHelper` described elsewhere in this book would need to be ported to use

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the SQLCipher for Android implementations of `SQLiteOpenHelper`, `SQLiteDatabase`, etc. This is not too difficult for an open source component like `SQLiteAssetHelper`.

Passwords and Sessions

Given an encrypted database, there are several ways that an attacker can try to access the data, including:

1. Use a brute-force attack via the app itself
2. Use a brute-force attack on the database directly, by copying it to some other machine
3. Obtain the password by the strategic deployment of a $5 wrench

The classic way to prevent the first approach is by having business logic that prevents lots of failed login attempts in a short period of time. This can be built into your login dialog (or the equivalent), tracking the number and times of failed logins and introducing delays, forced app exits, or something to add time and hassle for trying lots of passwords.

Since manually trying passwords is nasty, brutish, and long, many attackers would automate the process by copying the SQLCipher database to another machine (e.g., desktop) and running a brute-force attack on it directly. SQLCipher for Android has many built-in protections to help defend against this. So long as you are using a sufficiently long and complex encryption key, you should be fairly well-protected against such attacks.

Defending against wrenches is decidedly more difficult and is beyond the scope of this book.

About Those Passphrases…

Having a solid encryption algorithm, like the AES–256 used by default with SQLCipher for Android, is only half the battle. The other half is in using a high-quality passphrase, one that is unlikely to be guessed by anyone looking to break the encryption.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Upgrading to Encryption

Suppose you have an app already out on the market, and you decide that you want to add the option for encryption. It is fairly likely that the user will be miffed if they lose all their data in the process of switching to an encrypted database. Therefore, you will want to try to retain their data.

SQLCipher for Android does not support in-place encryption of database. However, it *does* support working with unencrypted databases and encrypted databases simultaneously, giving you the option of migration.

The approach boils down to:

- Open the unencrypted database in SQLCipher for Android, using an empty passphrase
- Use the `ATTACH` statement to open the encrypted database inside the same SQLCipher for Android session
- Use a supplied `sqlcipher_export()` function to migrate most of the data
- Copy the Android database schema version between the databases
- `DETACH` the encrypted database
- Close the unencrypted database (and, presumably, delete it)
- Use the encrypted database from this point forward

Since both database files will exist at one time, you will find it simplest to use separate names for them (e.g., `stuff.db` and `stuff-encrypted.db`).

To see how this works, take a look at the `Database/SQLCipherPassphrase` sample app (and the `Database/SQLCipherPassphrase-AndroidStudio` equivalent for Android Studio), which is a variation of the original, non-`ContentProvider` "constants" sample app, this time using SQLCipher for Android and supporting an upgrade from a non-encrypted database to an encrypted one.

The bulk of the logic for handling the encryption upgrade is in a static `encrypt()` method on our `DatabaseHelper`:

```
  static void encrypt(Context ctxt) {
    SQLiteDatabase.loadLibs(ctxt);

    File dbFile=ctxt.getDatabasePath(DATABASE_NAME);
    File legacyFile=ctxt.getDatabasePath(LEGACY_DATABASE_NAME);

    if (!dbFile.exists() && legacyFile.exists()) {
      SQLiteDatabase db=
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
          SQLiteDatabase.openOrCreateDatabase(legacyFile, "", null);

      db.rawExecSQL(String.format("ATTACH DATABASE '%s' AS encrypted KEY '%s';",
                                  dbFile.getAbsolutePath(), PASSPHRASE));
      db.rawExecSQL("SELECT sqlcipher_export('encrypted')");
      db.rawExecSQL("DETACH DATABASE encrypted;");

      int version=db.getVersion();

      db.close();

      db=SQLiteDatabase.openOrCreateDatabase(dbFile, PASSPHRASE, null);
      db.setVersion(version);
      db.close();

      legacyFile.delete();
    }
  }
```

First, we initialize SQLCipher for Android by calling `loadLibs()` on the SQLCipher version of `SQLiteDatabase`. We could do this someplace else, but for this sample, this is as good a spot as any.

We then create `File` objects pointing at the locations of the old, unencrypted database (with a name represented by a `LEGACY_DATABASE_NAME` static data member) and the new encrypted database (`DATABASE_NAME`). To get the `File` locations of those databases, we use `getDatabasePath()`, a method on `Context`, which returns the correct location for a database file given its name.

If the encrypted database exists, there is nothing that we need to do. Similarly, if it does not exist but the unencrypted database *also* does not exist, there is nothing that we *can* do. In either of those cases, we skip over the rest of the logic. In the first case, we already did the conversion (presumably); in the latter case, this is a new installation, and our `SQLiteOpenHelper` `onCreate()` logic will handle that. But, in the case where we do not have the encrypted database but do have the unencrypted one, we can create the encrypted database from the unencrypted data, which is what the bulk of the `encrypt()` method does.

To that, we:

- Use `openOrCreateDatabase()` to open the already-existing unencrypted database file *in SQLCipher for Android*, using `""` as the passphrase.
- Use a `rawExecSQL()` method available on the SQLCipher for Android version of `SQLiteDatabase` to `ATTACH` the encrypted database, given its path, to our database session, using the supplied passphrase. This means that we can

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

access the tables from both databases simultaneously, though we need to prefix all references to the attached database via its handle, `encrypted`.

- Use `rawExecSQL()` to execute `SELECT sqlcipher_export('encrypted')`, which copies most of our data from the unencrypted database (the database we have open) into the `encrypted` database (the one we attached). The big thing that `sqlcipher_export()` does *not* copy is the schema version number that Android maintains.
- Use `rawExecSQL()` to `DETACH` the attached `encrypted` database, as we no longer need it.
- Call `getVersion()` on the `SQLiteDatabase` representing the unencrypted database, to retrieve the schema version number that Android maintains.
- Close the unencrypted database and open the encrypted one using `openOrCreateDatabase()`.
- Use `setVersion()` on `SQLiteDatabase` to set the schema version of the encrypted database to the value we had from the unencrypted database.
- Close the encrypted database and delete the unencrypted database file. Note that on API Level 16+, we could use the `deleteDatabase()` method on `SQLiteDatabase` to cleanly delete everything associated with SQLite.

The combination of doing all of that migrates our data from an unencrypted database to an encrypted one.

Then, we simply need to call `encrypt()` before we try loading our constants, from `doInBackground()` of our `LoadCursorTask`:

```
    @Override
    protected Void doInBackground(Void... params) {
      DatabaseHelper.encrypt(ctxt);
      constantsCursor=doQuery();
      constantsCursor.getCount();

      return(null);
    }
```

To test this upgrade logic, you will need to:

- Run the original unencrypted version of this sample, found in the `Database/Constants` sample application
- Add a new constant using the unencrypted version of the app
- Run the encrypted version of the sample from this section, which shares the same package name as the original and therefore will replace it on your emulator

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You will see your added constant appear along with all of the standard ones, yet if you examine `/data/data/com.commonsware.android.constants/databases` on your ARM emulator via DDMS, you will see that your database is now named `constants-crypt.db` instead of `constants.db`, as we have replaced the unencrypted database with an encrypted one.

Changing Encryption Passphrases

Another thing the user might wish to do is change their passphrase. Perhaps they fear that their existing passphrase has been compromised (e.g., a narrow escape from a $5 wrench). Perhaps they rotate their passphrases as a matter of course. Perhaps they simply keep typing in their current one incorrectly and want to switch to one they think they can enter more accurately.

SQLCipher for Android supports a `rekey PRAGMA` that can accomplish this. Given an open encrypted database `db` — opened using the old passphrase – you can change the password to a `newPassword` string variable via:

```
db.execSQL(String.format("PRAGMA rekey = '%s'", newPassword));
```

Note that this may take some time, as SQLCipher for Android needs to re-encrypt the entire database.

Dealing with the Version 3.0.x Upgrade

If you are starting with SQLCipher for Android with the 3.0.x release, all is good.

If you have been using SQLCipher for Android from previous releases, but you are still in development mode, all is still good, so long as you can wipe out your old databases.

If you have apps in production using SQLCipher for Android from previous releases, you will have a small headache: the database structure has changed. SQLCipher for Android provides us with a `PRAGMA cipher_migrate` that we can run to upgrade the database in place to the new structure, once we have opened the database with our passphrase. However:

1. There is no great built-in place to put the code for calling this pragma
2. You do not want to blindly call this pragma every time you open the database, as it results in extra processing time

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

SQLCipher for Android, in an attempt to help with this, offers a modified version of methods like `openOrCreateDatabase()` on `SQLiteDatabase`, ones that take a `SQLiteDatabaseHook` implementation as the last parameter. This interface requires two methods:

1. `preKey()`, called after the database is opened but before the passphrase is applied
2. `postKey()`, called after the database is opened and after the passphrase is applied, but before anything else is done (e.g., standard `SQLiteOpenHelper` schema version checking)

Both methods are passed the `SQLiteDatabase` as a parameter, for you to do with as needed. So, for example, you could have a `postKey()` implementation that does the `postKey()` call only if needed:

```
public class SQLCipherV3Hook implements SQLiteDatabaseHook {
  private static final String PREFS=
      "net.sqlcipher.database.SQLCipherV3Helper";

  public static void resetMigrationFlag(Context ctxt, String dbPath) {
    SharedPreferences prefs=
        ctxt.getSharedPreferences(PREFS, Context.MODE_PRIVATE);
    prefs.edit().putBoolean(dbPath, false).commit();
  }

  @Override
  public void preKey(SQLiteDatabase database) {
    // no-op
  }

  @Override
  public void postKey(SQLiteDatabase database) {
    SharedPreferences prefs=
        getContext().getSharedPreferences(PREFS, Context.MODE_PRIVATE);
    boolean isMigrated=prefs.getBoolean(database.getPath(), false);

    if (!isMigrated) {
      database.rawExecSQL("PRAGMA cipher_migrate;");
      prefs.edit().putBoolean(database.getPath(), true).commit();
    }
  }
}
```

You can also pass a `SQLiteDatabaseHook` implementation into the `SQLiteOpenHelper` constructor as the fifth parameter, which will be used when `SQLiteOpenHelper` works with the underlying `SQLiteDatabase`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Multi-Factor Authentication

Another way to effectively boost the strength of your security is to implement your own multi-factor authentication. In this case, the passphrase is not obtained solely through the user typing in the whole thing, but instead is synthesized from two or more sources. So, in addition to some `EditText` widget for entering in a portion of the passphrase, the rest could come from things like:

- A value written to an NFC tag that the user must tap
- A value encoded in a QR code that the user must scan
- A value obtained by some Bluetooth-connected device via a custom protocol

You, in code, would concatenate the pieces together, possibly using delimiters that cannot be typed in (e.g., ASCII characters below 32) to denote the sources of each segment of the passphrase. The result would be the actual passphrase you would use with SQLCipher for Android.

The objective is to make it easier for users to have more complex passphrases, while not having to type in something complex every time. Tapping an NFC tag is much faster than tapping out a passphrase on a typical phone keyboard, for example. Also, the "something you know and something you have" benefit of multi-factor authentication can help with defending against $5 wrench attacks: if the NFC tag was destroyed, and the user never knew the portion of the passphrase stored on it, the user cannot divulge it.

Of course, this adds risks, such as the NFC tag being destroyed accidentally (e.g., "my dog ate it"). This can be mitigated in some cases by some "admin" being able to reset the password or supply a new NFC tag. In that case, getting the credentials requires *two* kidnappings and *two* $5 wrenches (or the serial application of a single $5 wrench, if budgets preclude buying two such wrenches), adding to the degree of difficulty for breaking the encryption by that means.

Detecting Failed Logins

If you try to decrypt a database using the incorrect passphrase — whether an attempt by outsiders to use the app, or the user "fat-fingering" the passphrase and making a typo — you will get an exception:

```
11-19 09:17:22.700: E/SQLiteOpenHelper(1634):
net.sqlcipher.database.SQLiteException: file is encrypted or is not a
database
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Alas, this is not a specific exception, making it a bit difficult to detect failed passphrases specifically. Your options are:

- Assume that your testing is sound and that exceptions when opening a database represent invalid passphrases, or
- Use a generic error message that hints at an invalid passphrase but leaves open the possibility of something else being wrong, or
- Read into the exception's message looking for "file is encrypted or is not a database", though this is fragile in the face of changes to SQLCipher for Android

SQLCipher for Android and Performance

Some developers worry about the overhead that encryption will place on the database I/O, and therefore worry that SQLCipher for Android will make their app unacceptably slow.

The impact of SQLCipher is not that bad, particularly for hardware with faster CPUs. Encryption is CPU-intensive, so faster CPUs reduce the overhead of the encryption. Also, since the disk I/O is comparable between SQLite and SQLCipher, the fact that flash memory is slow will mean that disk I/O, not decryption speed, will be the primary determinant of the speed of your queries. Similarly, disk I/O will count for more than CPU speed for the encryption needed for INSERT/UPDATE/DELETE operations.

For example, porting one relatively crude benchmark to use SQLCipher for Android showed no statistically significant performance difference from the SQLite edition on a Nexus 5 running Android 4.4.2.

To the extent that encryption adds overhead, it will tend to magnify existing problems. For example, anything that involves a "table scan" (i.e., a non-indexed lookup of database contents) will need more pages to be decrypted and, therefore, more decryption time. If your database I/O is well-tuned for SQLite, such as adding appropriate indexes, then your SQLCipher for Android overhead should be nominal.

Of course, the worse the CPU, the worse the story, and so older/cheaper devices may fare worse with SQLCipher for Android by comparison.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Encrypted Preferences

There are effectively three forms of data storage in Android:

- SQLite databases
- `SharedPreferences`
- Arbitrary files, in whatever format you want

You can encrypt SQLite via SQLCipher for Android, as seen in this chapter. You can encrypt arbitrary files as part of your data format, such as via `javax.crypto`.

What is not supported, out of the box, is a way to encrypt `SharedPreferences`.

There are two approaches for encrypting the contents of `SharedPreferences`:

1. Encrypt the container in which the `SharedPreferences` are stored
2. Encrypt each preference value as you store it in the `SharedPreferences`, and decrypt it when you read the value back out

Encryption via Custom SharedPreferences

`SharedPreferences` is an interface. Hence, you can create other implementations of that interface that store their data in something other than unencrypted XML files.

`CWSharedPreferences` is one such implementation. You can find it in the `cwac-prefs` project on GitHub.

`CWSharedPreferences` handles the `SharedPreferences` and `SharedPreferences.Editor` interfaces, along with the in-memory representations of the preferences. It then delegates the work of *storing* the preferences to a strategy object, implementing a strategy interface (`CWSharedPreferences.StorageStrategy`). Two such strategy implementations are supplied in the project: one using ordinary SQLite, and one using SQLCipher for Android.

The basic recipe for using `CWSharedPreferences` is:

- Create the strategy object, such as

```
new SQLCipherStrategy(getContext(), NAME, "atestpassword", LoadPolicy.SYNC)
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

(here, `NAME` is the name of the set of preferences, `"atestpassword"` is your passphrase, and `LoadPolicy.SYNC` indicates that the preferences should be loaded from disk immediately, not on a background thread)

- Create a `CWSharedPreferences` that employs your chosen strategy:

```
new CWSharedPreferences(yourStrategyObjectGoesHere);
```

- Use the `CWSharedPreferences` as you would any other `SharedPreferences` implementation
- Call `close()` on the strategy object, to release any resources that it might hold (e.g., open database connection)

Encryption via Custom Preference UI and Accessors

The big drawback to the custom `SharedPreferences` is the fact that you cannot get the `PreferenceScreen` system to work with it. The preference UI is hard-wired to use the stock implementation of `SharedPreferences` and does not appear to support any way to substitute in some other implementation.

Hence, another approach is to keep things in standard `SharedPreferences`' XML files, but encrypt text values on a preference-by-preference basis. Since the data type needs to remain the same, most likely you would restrict this to encrypting strings (e.g., `EditTextPreference`, `ListPreference`) rather than numbers, booleans, etc.

To do this, you would need to:

- Implement static methods somewhere for your encryption and decryption algorithms
- Subclass the `Preference` classes of interest and override methods that would deal with the raw preference data, like `onDialogClosed()`, to encrypt the values you persist and decrypt the values you read in, using the static methods mentioned above
- Use your extended `Preference` classes in your preference XML as needed
- Use those static methods as part of reading (or writing) the preference values directly via `SharedPreferences`

The downsides to this approach include:

- Only certain preferences are encrypted, rather than all of them

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- You lose some of the low-level encryption power of SQLCipher for Android, such as automatic hashing of passphrases, which you would have to handle yourself
- There may not be a library that supplies these extended `Preference` classes, forcing you to roll your own

IOCipher

SQLCipher for Android is also used as the backing store for IOCipher. IOCipher is a virtual file system (VFS) for Android, allowing you to write code that looks and works like it uses normal file I/O, yet all of the files are actually saved as BLOBs in a SQLCipher for Android database. The result is a fully-encrypted VFS, inheriting all of SQLCipher's security features, such as default AES–256 encryption. This may be easier for you to use than encrypting and decrypting files individually via `javax.crypto`, for example.

IOCipher is considered to be in pre-alpha state as of November 2012.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Packaging and Distributing Data

Sometimes, you not only want to ship your code and simple resources with your app, but you also want to ship other types of data, such as an initial database that your app will use when first run. This chapter will examine the means by which you can do those sorts of things.

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- database access
- content provider theory
- content provider implementations

Packing a Database To Go

Android's support for databases is focused on databases you create and populate entirely at runtime. Even if you want some initial data in the database, the expectation is that you would add that via Java code, such as the series of `insert()` calls we made in the `DatabaseHelper` of the various flavors of the `ConstantsBrowser` sample application.

However, that is tedious and slow for larger initial data sets, even if you make careful use of transactions to minimize the disk I/O.

What would be nice is to be able to ship a pre-populated database with your app. While Android does not offer built-in support for this, there are a few ways you can accomplish it yourself. One of the easiest, though, is to use existing third-party code

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

that supports this pattern, such as Jeff Gilfelt's `SQLiteAssetHelper`, available via a GitHub repository.

Eclipse users can download a JAR. Android Studio users can add a `compile` statement to the `dependencies` closure in `build.gradle` to pull in `com.readystatesoftware.sqliteasset:sqliteassethelper:...` (for some version indicated by `...`) from `mavenCentral()`.

`SQLiteAssetHelper` replaces your existing `SQLiteOpenHelper` subclass with one that handles database creation and upgrading for you. Rather than you writing a lot of SQL code for each of those, you provide a pre-populated SQLite database (for creation) and a series of SQL scripts (for upgrades). `SQLiteAssetHelper` then does the work to set up your pre-populated database when the database is first accessed and running your SQL scripts as needed to handle schema changes. And, `SQLiteAssetHelper` is open source, licensed under the same Apache License 2.0 that is used for Android proper.

To examine `SQLiteAssetHelper` in action, let's look at the `Database/ConstantsAssets` sample project. This is yet another rendition of the same app as the other flavors of `ConstantsBrowser`, but one where we use a pre-populated database.

Create and Pack the Database

Whereas normally you create your SQLite database at runtime from Java code in your app, you now create your SQLite database using whatever tools you like, at development time. Whether you use the command-line `sqlite3` utility, the SQLite Manager extension for Firefox, or anything else, is up to you. You will need to set up all of your tables, indexes, and so forth.

Then, you need to:

1. Create an `assets/databases/` directory in your project
2. Copy your database into this directory (or put it there in the first place, if you prefer)

If your `minSdkVersion` is less than 11, you will instead need to have a ZIP or GZIP archive containing the database. The archive should have the same name as the database file, just with the `.zip` or `.gz` extension. The reason for the ZIP compression comes from an Android 1.x/2.x limitation – assets that are compressed by the Android build tools have a file-size limitation (around 1MB). Hence, you need

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

to store larger files in a file format that will not be compressed by the Android build tools, and those tools will not try to compress a `.zip` file.

In the `ConstantsAssets` project, you will see an `assets/databases/constants.db` file, containing a copy of the SQLite database with our `constants` table and pre-populated values.

Unpack the Database, With a Little Help(er)

Your compressed database will ship with your APK. To get it into its regular position on internal storage, you use `SQLiteAssetHelper`. Simply create a subclass of `SQLiteAssetHelper` and override its constructor, supplying the same values as you would for a `SQLiteOpenHelper` subclass, notably the database name and schema revision number. Note that the database name that you use must match the filename of the compressed database, minus the `.zip` extension.

So, for example, our new `DatabaseHelper` looks like this:

```
package com.commonsware.android.dbasset;

import android.content.Context;
import com.readystatesoftware.sqliteasset.SQLiteAssetHelper;

class DatabaseHelper extends SQLiteAssetHelper {
  static final String TITLE="title";
  static final String VALUE="value";
  static final String TABLE="constants";
  private static final String DATABASE_NAME="constants.db";

  public DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, 1);
  }
}
```

`SQLiteAssetHelper` will then copy your database out of assets and set it up for conventional use, as soon as you call `getReadableDatabase()` or `getWritableDatabase()` on an instance of your `SQLiteAssetHelper` subclass.

Upgrading Sans Java

Traditionally, with `SQLiteOpenHelper`, to handle a revision in your schema, you override `onUpgrade()` and do the upgrade work in there. With `SQLiteAssetHelper`, there is a built-in `onUpgrade()` method that uses SQL scripts in your APK to do the upgrade work instead.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

These scripts will also reside in your `assets/databases/` directory of your project. The name of the file will be `$NAME_upgrade_$FROM-$TO.sql`, where you replace `$NAME` with the name of your database (e.g., `constants.db`), `$FROM` with the old schema version number (e.g., `1`) and `$TO` with the new schema version number (e.g., `2`). Hence, you wind up with files like `assets/databases/constants.db_upgrade_1-2.sql`. This should contain the SQL statements necessary to upgrade your schema between the versions.

`SQLiteAssetHelper` will chain these together as needed. Hence, to upgrade from schema version 1 to 3, you could either have a single dedicated 1->3 script, or a 1->2 script and a 2->3 script.

Limitations

The biggest limitation comes with disk space. Since APK files are read-only at runtime, you cannot delete the copy of the database held as an asset in your APK file once `SQLiteAssetHelper` has unpacked it. This means that the space taken up by your ZIP file will be taken up indefinitely. Note, though, that you could use this to your advantage, offering the user a "start over from scratch" option that deletes their existing database, so `SQLiteAssetHelper` will unpack a fresh original copy on the next run. Or, you could implement a `SQLiteDownloadHelper` that follows the `SQLiteAssetHelper` approach but obtains its database from the Internet instead of from assets.

In principle, SQLite could change their file format. If that ever happens, you will need to make sure that you create a SQLite database in the file format that can be used by Android, more so than what can be used by the latest SQLite standalone tools.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Advanced Database Techniques

This chapter offers tips and techniques for working with SQLite beyond what the previous chapters in the book have covered.

Prerequisites

This chapter assumes that you have read the core chapters, particularly the ones on databases and Internet access.

Also, please read the chapter on advanced action bar techniques, particularly the section on `SearchView`, as that is used in one of the sample apps.

Full-Text Indexing

Standard SQL databases are great for ordinary queries. In particular, when it comes to text, SQL databases are great for finding rows where a certain column value matches a particular string. They are usually pretty good about finding when a column value matches a particular string prefix, if there is an index on that column. Things start to break down when you want to search for an occurrence of a string in a column, as this usually requires a "table scan" (i.e., iteratively examining each row to see if this matches). And getting more complex than that is often impossible, or at least rather difficult.

SQLite, in its stock form, inherits all those capabilities and limitations. However, SQLite also offers full-text indexing, where we can search our database much like how we use a search engine (e.g., "find all rows where this column has both `foo` and `bar` in it somewhere"). While a full-text index takes up additional disk space, the speed of the full-text searching is quite impressive.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For example, if you are reading this book using the Android APK edition (instead of the PDF, EPUB, or Kindle/MOBI editions), tap on the `SearchView` action bar item and search for `FTS4`. You will get a list of matches back almost instantaneously, despite the fact that you are searching a multi-megabyte book. That is because this book ships a SQLite-powered full-text index of the book's contents, specifically to power your use of `SearchView`.

In this section, we will review how you can add full-text indexing to your SQLite database and how you can let the user take advantage of that index using a `SearchView`.

First, a Word About SQLite Versions

SQLite has evolved since Android's initial production release in 2008.

In many cases, Android does not incorporate updates to third-party code, for backwards-compatibility reasons (e.g., Apache's HttpClient). In the case of SQLite, newer Android versions do take on newer versions of SQLite... but the exact version of SQLite that a given version of Android uses is undocumented. Worse, some device manufacturers replace the stock SQLite for a version of Android with a different one.

This Stack Overflow answer contains a mapping of Android OS releases to SQLite versions, including various "anomalies" where manufacturers have elected to ship something else.

In many cases, the SQLite version does not matter. Core SQLite capabilities will have existed since the earliest days of Android. However, full-text indexing did not exist in the first SQLite used by Android, meaning that you will have to pay attention to your `minSdkVersion` and aim high enough that devices should support the full-text indexing option you choose.

Note that you could use an external SQLite implementation, one that gives you a newer SQLite engine than what might be on the device. For example, SQLCipher for Android ships its own copy of SQLite (with the SQLCipher extensions compiled in), one that is often newer than the one that is baked into the firmware of any given device.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

FTS3 and FTS4

There are two full-text indexing options available in SQLite: FTS3 and FTS4. FTS4 can be much faster on certain queries, though overall the speed of the two implementations should be similar. FTS4 has two key limitations:

1. It may take a bit more disk space for its indexes.
2. It was added to SQLite 3.7.4, which was only introduced into standard Android in API Level 11.

The sample app for this section will demonstrate FTS4, as that is available on most Android devices.

Note that the Android developer documentation does not cover FTS3 or FTS4 full-text indexing. The details for the SQL syntax to support these options can be found in the SQLite documentation.

Creating a Full-Text Indexed Table

A full-text indexed table, using FTS3 or FTS4, uses SQLite's `CREATE VIRTUAL TABLE` syntax. This indicates that you are opting into some special table-storage behavior, rather than the stock stuff.

In the `Database/FTS` sample project, the `onCreate()` method of our `SQLiteOpenHelper` subclass (`DatabaseHelper`) creates such a virtual table, using FTS4 for full-text indexing:

```
@Override
public void onCreate(SQLiteDatabase db) {
  db.execSQL("CREATE VIRTUAL TABLE questions USING fts4("
                +"_id INTEGER PRIMARY KEY, title TEXT, "
                +"link TEXT, profileImage TEXT, creationDate INTEGER, "
                +"order=DESC);");
}
```

There are a few differences here from a typical `CREATE TABLE` statement, beyond the introduction of the `VIRTUAL` keyword:

- The `USING fts4` indicates that the virtual table is employing the FTS4 full-text indexing engine. To use FTS3, just replace `fts4` with `fts3`.
- You can have key-value pairs in the column list, separated by equals signs, to provide options for configuring the virtual table. In this case, it will provide

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

options for configuring the FTS4 indexing behavior. In this case, we are providing `order=DESC`, to indicate that the full-text index should be optimized for returning items in descending order. Note that these options only exist for FTS4, not FTS3. The full roster of available options is covered in the SQLite documentation.

This gives us a table that supports normal table operations but also has a full-text index for its columns. However, there are some limitations, notably that these tables ignore constraints. So, for example, the `PRIMARY KEY` constraint applied to the `_id` column is ignored.

Populating a Full-Text Indexed Table

Adding content to an FTS3 or FTS4 table uses the same `INSERT` statements that you might use for a regular table. For example, the `DatabaseHelper` in the sample app has an `insertQuestions()` method that deletes all existing rows in the `questions` table, then inserts a bunch of rows based on a supplied `List` of `Item` objects:

```
void insertQuestions(Context app, List<Item> items) {
  SQLiteDatabase db=getDb(app);

  db.beginTransaction();

  db.delete("questions", null, null);

  try {
    for (Item item : items) {
      Object[] args={ item.id, item.title, item.link,
                      item.owner.profileImage, item.creationDate};

      db.execSQL("INSERT INTO questions (_id, title, "
                   +"link, profileImage, creationDate) "
                   +"VALUES (?, ?, ?, ?, ?)",
                 args);
    }

    db.setTransactionSuccessful();
  }
  finally {
    db.endTransaction();
  }
}
```

If those `Item` objects look familiar, that is because this app is a modified version of the Stack Overflow questions apps profiled in the chapter on Internet access.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The reason why we are deleting everything before inserting is just to keep the sample simple. The database table will hold all of the questions pulled from the Stack Exchange API. Each time we run the app, we get the latest questions from that API. The vision was to use `INSERT OR REPLACE` or `INSERT OR IGNORE` statements to be able to merge content into the table. However, FTS3 and FTS4 tables ignore all constraints, as noted above, which prevents the conflict resolution options (e.g., `OR REPLACE`) from working. Hence, rather than manually sifting through to find if there is an existing row or not for a given ID value, this sample simply gets rid of all existing rows. A production-grade app would likely apply a more sophisticated algorithm.

Querying a Full-Text Indexed Table

While you can query a full-text indexed table using normal `SELECT` statements, usually the point is to apply the `MATCH` operator, as is seen in the `loadQuestions()` method from `DatabaseHelper`:

```
Cursor loadQuestions(Context app, String match) {
  SQLiteDatabase db=getDb(app);

  if (TextUtils.isEmpty(match)) {
    return(db.rawQuery("SELECT * FROM questions ORDER BY creationDate DESC",
                       null));
  }

  String[] args={ match };

  return(db.rawQuery("SELECT * FROM questions WHERE title "
                     +"MATCH ? ORDER BY creationDate DESC", args));
}
```

The `MATCH` operator supports a wide range of query structures, including:

- Keyword matches (e.g., `Android`)
- Prefix matches (e.g., `SQL*`)
- Phrase matches (e.g., `"open source"`)
- `NEAR`, `AND`, `OR`, and `NOT` operators (e.g., `sqlite AND database`)

The result is the same sort of `Cursor` that you would get from a regular `SELECT` statement against a non-full-text-indexed table.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Some Notes About the Rest of the Sample App

As noted previously, this sample app is a revised version of the Stack Overflow questions list from the chapter on Internet access. It is specifically derived from the Picasso version of the sample. However, this version is designed to allow the user to full-text search the downloaded question data (e.g., title), above and beyond just seeing the list of latest questions.

This, in turn, requires a few more changes than those outlined so far. The following sections outline some of the highlights.

Adding a ModelFragment

The original sample had a very simple data model: a list of questions retrieved via Retrofit. Hence, the sample did not include much in the way of model management.

The FTS sample needs a database, which implies more local disk I/O that we are responsible for, which in turn leads us in the direction of implementing a model fragment (`ModelFragment`), much as the tutorials and a few other samples do:

```
package com.commonsware.android.fts;

import android.app.Activity;
import android.app.Fragment;
import android.content.Context;
import android.database.Cursor;
import android.os.Bundle;
import android.util.Log;
import de.greenrobot.event.EventBus;
import retrofit.RestAdapter;

public class ModelFragment extends Fragment {
  private Context app=null;

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    setRetainInstance(true);
  }

  @Override
  public void onAttach(Activity host) {
    super.onAttach(host);

    EventBus.getDefault().register(this);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (app==null) {
      app=host.getApplicationContext();
      new FetchQuestionsThread().start();
    }
  }

  @Override
  public void onDetach() {
    EventBus.getDefault().unregister(this);

    super.onDetach();
  }

  public void onEventBackgroundThread(SearchRequestedEvent event) {
    try {
      Cursor results=DatabaseHelper.getInstance(app).loadQuestions(app,
event.match);

      EventBus.getDefault().postSticky(new ModelLoadedEvent(results));
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(),
          "Exception searching database", e);
    }
  }

  class FetchQuestionsThread extends Thread {
    @Override
    public void run() {
      RestAdapter restAdapter=
          new RestAdapter.Builder().setEndpoint("https://api.stackexchange.com")
              .build();
      StackOverflowInterface so=
          restAdapter.create(StackOverflowInterface.class);

      SOQuestions questions=so.questions("android");

      try {
        DatabaseHelper
            .getInstance(app)
            .insertQuestions(app, questions.items);
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(),
            "Exception populating database", e);
      }

      try {
        Cursor results=DatabaseHelper.getInstance(app).loadQuestions(app, null);

        EventBus.getDefault().postSticky(new ModelLoadedEvent(results));
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(),
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
            "Exception populating database", e);
      }
    }
  }
}
```

In `onCreate()`, we mark this fragment as retained, as that is key to the model fragment pattern, so the fragment retains the model data across configuration changes.

In `onAttach()`, we register for the greenrobot EventBus, plus kick off a `FetchQuestionsThread` if we have not done so already (i.e., this is the first `onAttach()` call we have received). `onDetach()` unregisters us from the event bus.

`FetchQuestionsThread`, in turn, uses Retrofit to download the questions from Stack Overflow, then uses `DatabaseHelper` to insert the questions into the FTS-enabled database table, then uses the `DatabaseHelper` again to retrieve all existing questions in the form of a `Cursor`, which it wraps in a `ModelLoadedEvent` and posts to the `EventBus`. This time, though, it posts it as a sticky event.

That sticky event is consumed by a revised version of the `QuestionsFragment`, in its `onEventMainThread()` method:

```
  public void onEventMainThread(ModelLoadedEvent event) {
    ((SimpleCursorAdapter)getListAdapter()).changeCursor(event.model);

    if (sv!=null) {
      sv.setEnabled(true);
    }
  }
```

But because this is a sticky event, we will get this event both when it is raised (because the data is loaded) and any time thereafter when the fragment registers with the `EventBus`. This allows `QuestionsFragment` to *not* be retained, as it will get back the bulk of its model data automatically from greenrobot's EventBus.

`QuestionsFragment` also is modified from the Picasso sample to deal with the fact that its model data is now a `Cursor`, so it uses `SimpleCursorAdapter` to populate the list. To handle loading avatar images from the URLs, `QuestionsFragment` adds a `QuestionBinder` implementation of `ViewBinder` to the `SimpleCursorAdapter`, where `QuestionBinder` handles the Picasso logic from before:

```
  private class QuestionBinder implements SimpleCursorAdapter.ViewBinder {
    int size;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    QuestionBinder() {
      size=getActivity()
              .getResources()
              .getDimensionPixelSize(R.dimen.icon);
    }

    @Override
    public boolean setViewValue (View view, Cursor cursor, int columnIndex) {
      switch (view.getId()) {
        case R.id.title:
          ((TextView)view).setText(Html.fromHtml(cursor.getString(columnIndex)));

          return(true);

        case R.id.icon:
          Picasso.with(getActivity()).load(cursor.getString(columnIndex))
              .resize(size, size).centerCrop()
              .placeholder(R.drawable.owner_placeholder)
              .error(R.drawable.owner_error).into((ImageView)view);

          return(true);
      }

      return(false);
    }
  }
```

The main activity (`MainActivity`) sets up the `ModelFragment` in `onCreate()`, at least when one does not already exist due to a configuration change:

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new QuestionsFragment()).commit();
    }

    model=(ModelFragment)getFragmentManager().findFragmentByTag(MODEL);

    if (model==null) {
      model=new ModelFragment();
      getFragmentManager().beginTransaction().add(model, MODEL).commit();
    }
  }
```

This description, though, has skipped over the `onEventBackgroundThread()` method on the `ModelFragment`, which we will get to later in this overview.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Adding a SearchView

As is covered in the chapter on advanced action bar techniques, a `SearchView` can be used to provide the standard "magnifying glass" search icon in the action bar. When tapped, the action bar item expands into a field where the user can type something, which our code can then receive and use to update the UI. In the `SearchView` sample from the action bar chapter, we saw using a `SearchView` for filtering. This time, we will use a `SearchView` for searching.

For a search, we need to know when the user is done typing, which is usually done by the user clicking a submit button. Hence, our code to configure the `SearchView` (a `configureSearchView()` method in `QuestionsFragment`) calls `setSubmitButtonEnabled(true)`:

```
  private void configureSearchView(Menu menu) {
    MenuItem search=menu.findItem(R.id.search);

    search.setOnActionExpandListener(this);
    sv=(SearchView)search.getActionView();
    sv.setOnQueryTextListener(this);
    sv.setSubmitButtonEnabled(true);
    sv.setIconifiedByDefault(true);

    if (initialQuery != null) {
      sv.setIconified(false);
      search.expandActionView();
      sv.setQuery(initialQuery, true);
    }
  }
```

This, in turn, means that we need to pay attention to `onQueryTextSubmit()` in our `SearchView.OnQueryTextListener` implementation. That interface is implemented on `QuestionsFragment` itself, and delegates its work to a `doSearch()` method:

```
  @Override
  public boolean onQueryTextSubmit(String query) {
    doSearch(query);

    return(true);
  }
```

That method, in turn, confirms that the search is different than the last one we did (so we do not waste time running the search again), disables the `SearchView`, and posts a `SearchRequestedEvent` on the `EventBus` with the user's search string:

```
  private void doSearch(String match) {
    if (!match.equals(lastQuery)) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      lastQuery=match;

      if (sv != null) {
        sv.setEnabled(false);
      }

      EventBus.getDefault().post(new SearchRequestedEvent(match));
    }
  }
```

That event is picked up by `onEventBackgroundThread()` on `ModelFragment`. The method name `onEventBackgroundThread()` means that the event will be delivered to us on an EventBus-supplied background thread, so we can perform database I/O. In it, we call `loadQuestions()` on the `DatabaseHelper` to perform the search, and post another sticky `ModelLoadedEvent` to update the UI with the search results and re-enable the `SearchView`:

```
  public void onEventBackgroundThread(SearchRequestedEvent event) {
    try {
      Cursor results=DatabaseHelper.getInstance(app).loadQuestions(app,
event.match);

      EventBus.getDefault().postSticky(new ModelLoadedEvent(results));
    }
    catch (Exception e) {
      Log.e(getClass().getSimpleName(),
          "Exception searching database", e);
    }
  }
```

When the user clears the `SearchView`, such as by pressing the BACK button a few times, the `onMenuItemActionCollapse()` method of `QuestionsFragment` calls a `clearSearch()` method:

```
  @Override
  public boolean onMenuItemActionCollapse(MenuItem item) {
    clearSearch();

    return(true);
  }
```

That `clearSearch()` method simply posts another `SearchRequestedEvent`, this time to load a fresh roster of all questions:

```
  private void clearSearch() {
    if (lastQuery!=null) {
      lastQuery=null;

      sv.setEnabled(false);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      EventBus.getDefault().post(new SearchRequestedEvent(null));
    }
  }
```

The Results

When you run the app, you are initially presented with the list of questions pulled from the Stack Exchange API:

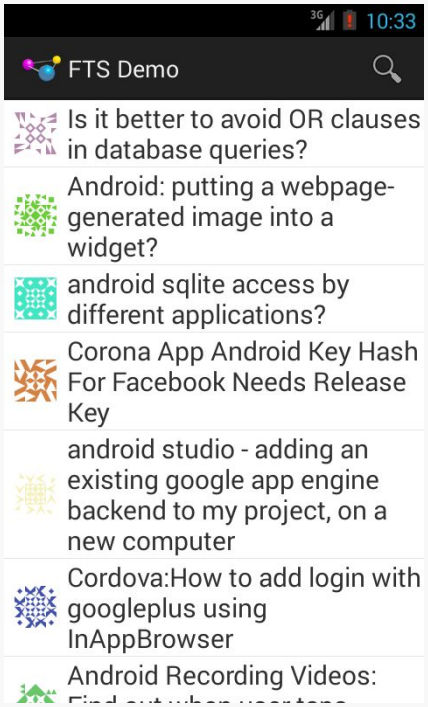


Figure 604: FTS Demo, As Initially Launched

Tapping on the `SearchView` opens it up, as normal, though this time with the "submit" button (the rightward-pointing arrowhead):

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

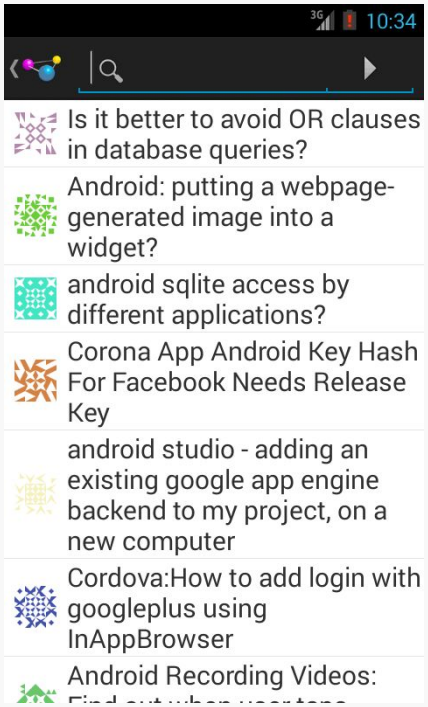


Figure 605: FTS Demo, with Open SearchView

Typing in a search, then tapping the "submit" button, will reload the list with those questions that match the search criteria in the question title:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

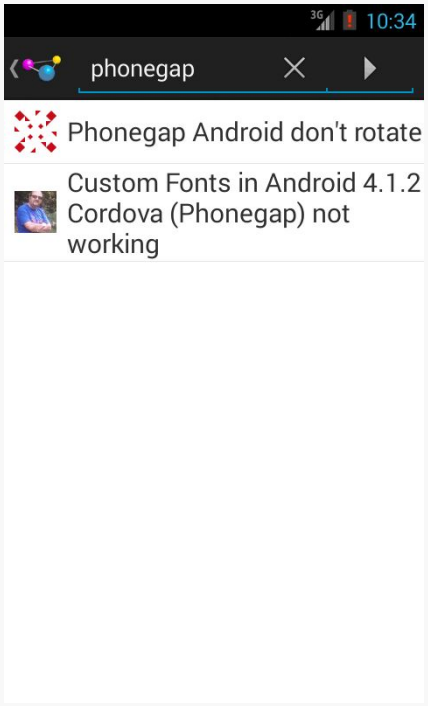


Figure 606: FTS Demo, Showing Basic Search

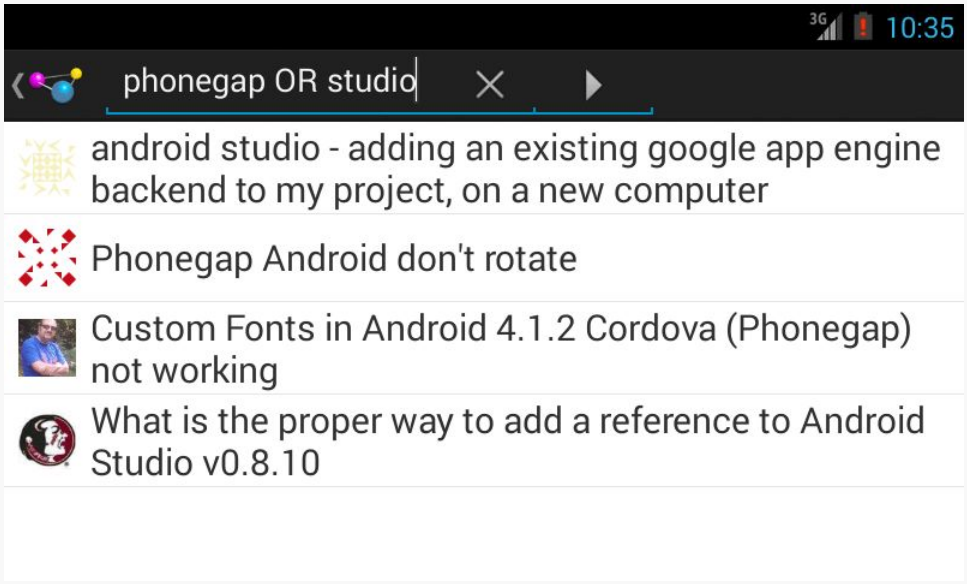


Figure 607: FTS Demo, Showing Boolean Search

Using the BACK button to get out of the `SearchView` reloads the full list of questions.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Getting Snippets

Usually, the content that is being indexed is a lot longer than Stack Overflow question titles. For example, it might be chapters in a book on Android application development. In that case, it would be useful to not only find out what chapters match the search expression, but what the prose is around the search expression, to help the user determine which search results are likely to be useful.

The APK edition of this book stores each paragraph and bullet as a separate entry in a SQLite database in an FTS3-enabled table. The query used when the reader types in a search expression in the app's `SearchView` is:

```
SELECT ROWID as _id, file, node, snippet(booksearch) AS snippet FROM
booksearch WHERE prose MATCH ?
```

Here, `file` and `node` are used to identify where this passage came from within the book, so when the user taps on a search result in the list, the book reader can jump to that particular location.

The `snippet()` auxiliary function will return, as the name suggests, a snippet of the indexed text, with the search match highlighted. It takes the name of the table `booksearch` as a mandatory parameter. It also supports optional parameters for what to bracket the search match with (defaults to `<b>` and `</b>`) and what to use for an ellipsis for extended prose segments (defaults to `<b>...</b>`). In the case of this query, the default formatting of the result is used. The resulting text can then be fed into `Html.fromHtml()` to generate the text for the `ListView` row, showing the search match within the snippet highlighted in bold:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

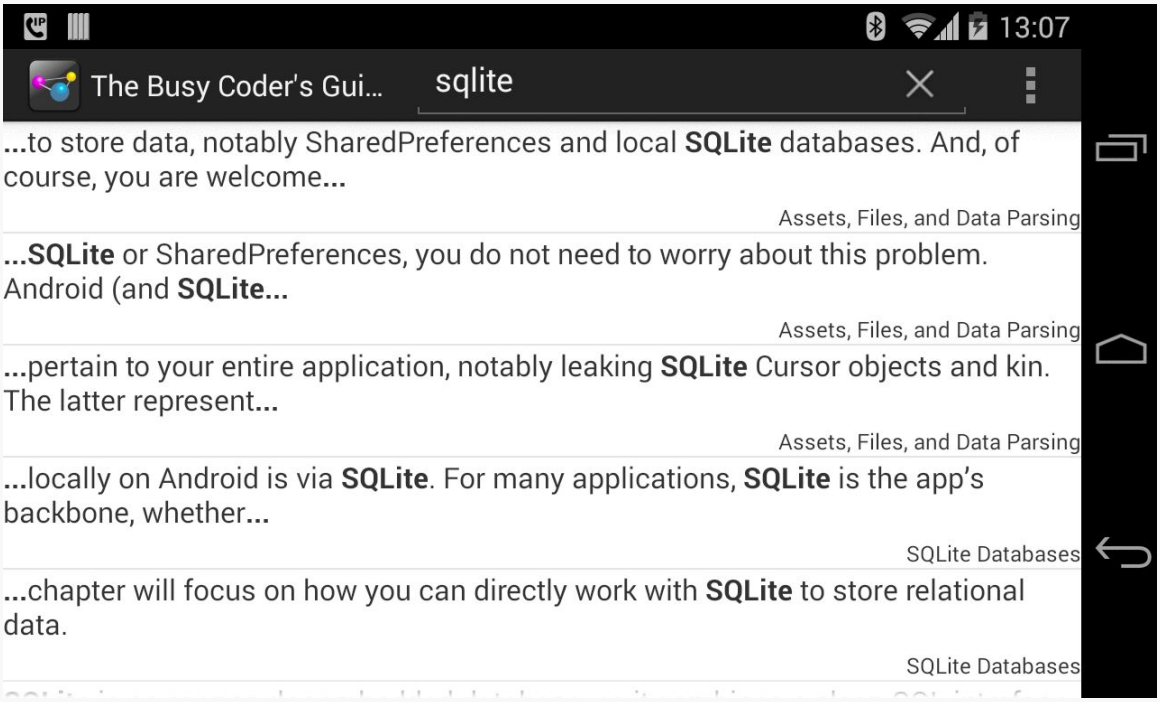


Figure 608: This Book's Reader App, Showing Search Results

The app also shows the name of the chapter in the lower-right corner of each row, to help provide larger context for where this snippet comes from.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Miscellaneous Network Capabilities

This chapter is a catch-all for various Android capabilities related to network I/O and the Internet, beyond what is covered elsewhere in the book.

(yes, this chapter could have a more exciting rationale for existing, but the author is subject to "Truth in Advertising" laws...)

Prerequisites

Readers of this chapter should have read the core chapters of the book.

Downloading Files

Android 2.3 introduced a `DownloadManager`, designed to handle a lot of the complexities of downloading larger files, such as:

1. Determining whether the user is on WiFi or mobile data, and if so, whether the download should occur
2. Handling when the user, previously on WiFi, moves out of range of the access point and "fails over" to mobile data
3. Ensuring the device stays awake while the download proceeds

`DownloadManager` itself is less complicated than the alternative of writing all of that stuff yourself. However, it does present a few challenges. In this section, we will examine the `Internet/Download` sample project, one that uses `DownloadManager`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Permissions

To use `DownloadManager`, you will need to hold the `INTERNET` permission. You will also need the `WRITE_EXTERNAL_STORAGE` permission, as `DownloadManager` can only download to external storage.

For example, here is the manifest for the `Internet/Download` application, where we request these two permissions:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.downmgr"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:anyDensity="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"/>

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="14"/>

  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Holo.Light.DarkActionBar">
    <activity
      android:name=".DownloadDemo"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

The Layout

Our sample application has a simple layout, consisting of three buttons:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. One to kick off a download
2. One to query the status of a download
3. One to display a system-supplied activity containing the roster of downloaded files

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <Button
    android:id="@+id/start"
    android:text="@string/start_download"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
  />
  <Button
    android:id="@+id/query"
    android:text="@string/query_status"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
    android:enabled="false"
  />
  <Button android:id="@+id/view"
    android:text="@string/view_log"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
  />
</LinearLayout>
```

Requesting the Download

To kick off a download, we first need to get access to the `DownloadManager`. This is a so-called "system service". You can call `getSystemService()` on any activity (or other `Context`), provide it the identifier of the system service you want, and receive the system service object back. However, since `getSystemService()` supports a wide range of these objects, you need to cast it to the proper type for the service you requested.

So, for example, here is the `onCreateView()` method of the `DownloadFragment`, in which we get the `DownloadManager`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    mgr=

(DownloadManager)getActivity().getSystemService(Context.DOWNLOAD_SERVICE);

    View result=inflater.inflate(R.layout.main, parent, false);

    query=result.findViewById(R.id.query);
    query.setOnClickListener(this);
    start=result.findViewById(R.id.start);
    start.setOnClickListener(this);

    result.findViewById(R.id.view).setOnClickListener(this);

    return(result);
  }
```

Most of these managers have no `close()` or `release()` or `goAwayPlease()` sort of methods — you can just use them and let garbage collection take care of cleaning them up.

Given the manager, we can now call an `enqueue()` method to request a download. The name is relevant — do not assume that your download will begin immediately, though often times it will. The `enqueue()` method takes a `DownloadManager.Request` object as a parameter. The `Request` object uses the builder pattern, in that most methods return the `Request` itself, so you can chain a series of calls together with less typing.

For example, the top-most button in our layout is tied to a `startDownload()` method in `DownloadFragment`, shown below:

```
  private void startDownload(View v) {
    Uri uri=Uri.parse("http://commonsware.com/misc/test.mp4");


Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS)
               .mkdirs();

    DownloadManager.Request req=new DownloadManager.Request(uri);

    req.setAllowedNetworkTypes(DownloadManager.Request.NETWORK_WIFI
                               | DownloadManager.Request.NETWORK_MOBILE)
       .setAllowedOverRoaming(false)
       .setTitle("Demo")
       .setDescription("Something useful. No, really.")
       .setDestinationInExternalPublicDir(Environment.DIRECTORY_DOWNLOADS,
                                          "test.mp4");
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    lastDownload=mgr.enqueue(req);
```

We are downloading a sample MP4 file, and we want to download it to the external storage area. To do the latter, we are using `getExternalStoragePublicDirectory()` on `Environment`, which gives us a directory suitable for storing a certain class of content. In this case, we are going to store the download in the `Environment.DIRECTORY_DOWNLOADS`, though we could just as easily have chosen `Environment.DIRECTORY_MOVIES`, since we are downloading a video clip. Note that the `File` object returned by `getExternalStoragePublicDirectory()` may point to a not-yet-created directory, which is why we call `mkdirs()` on it, to ensure the directory exists.

We then create the `DownloadManager.Request` object, with the following attributes:

1. We are downloading the specific URL we want, courtesy of the `Uri` supplied to the `Request` constructor
2. We are willing to use either mobile data or WiFi for the download (`setAllowedNetworkTypes()`), but we do not want the download to incur roaming charges (`setAllowedOverRoaming()`)
3. We want the file downloaded as `test.mp4` in the downloads area on the external storage (`setDestinationInExternalPublicDir()`)

We also provide a name (`setTitle()`) and description (`setDescription()`), which are used as part of the notification drawer entry for this download. The user will see these when they slide down the drawer while the download is progressing.

The `enqueue()` method returns an ID of this download, which we hold onto for use in querying the download status.

Keeping Track of Download Status

If the user presses the Query Status button, we want to find out the details of how the download is progressing. To do that, we can call `query()` on the `DownloadManager`. The `query()` method takes a `DownloadManager.Query` object, describing what download(s) you are interested in. In our case, we use the value we got from the `enqueue()` method when the user requested the download:

```
  private void queryStatus(View v) {
    Cursor c=
        mgr.query(new DownloadManager.Query().setFilterById(lastDownload));

    if (c == null) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      Toast.makeText(getActivity(), R.string.download_not_found,
                     Toast.LENGTH_LONG).show();
    }
    else {
      c.moveToFirst();

      Log.d(getClass().getName(),
            "COLUMN_ID: "
                + c.getLong(c.getColumnIndex(DownloadManager.COLUMN_ID)));
      Log.d(getClass().getName(),
            "COLUMN_BYTES_DOWNLOADED_SO_FAR: "
                +
c.getLong(c.getColumnIndex(DownloadManager.COLUMN_BYTES_DOWNLOADED_SO_FAR)));
      Log.d(getClass().getName(),
            "COLUMN_LAST_MODIFIED_TIMESTAMP: "
                +
c.getLong(c.getColumnIndex(DownloadManager.COLUMN_LAST_MODIFIED_TIMESTAMP)));
      Log.d(getClass().getName(),
            "COLUMN_LOCAL_URI: "
                +
c.getString(c.getColumnIndex(DownloadManager.COLUMN_LOCAL_URI)));
      Log.d(getClass().getName(),
            "COLUMN_STATUS: "
                + c.getInt(c.getColumnIndex(DownloadManager.COLUMN_STATUS)));
      Log.d(getClass().getName(),
            "COLUMN_REASON: "
                + c.getInt(c.getColumnIndex(DownloadManager.COLUMN_REASON)));

      Toast.makeText(getActivity(), statusMessage(c), Toast.LENGTH_LONG)
           .show();

      c.close();
    }
  }
```

The `query()` method returns a `Cursor`, containing a series of columns representing the details about our download. There is a series of constants on the `DownloadManager` class outlining what is possible. In our case, we retrieve (and dump to LogCat):

1. The ID of the download (`COLUMN_ID`)
2. The amount of data that has been downloaded to date (`COLUMN_BYTES_DOWNLOADED_SO_FAR`)
3. What the last-modified timestamp is on the download (`COLUMN_LAST_MODIFIED_TIMESTAMP`)
4. Where the file is being saved to locally (`COLUMN_LOCAL_URI`)
5. What the actual status is (`COLUMN_STATUS`)
6. What the reason is for that status (`COLUMN_REASON`)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that COLUMN_LOCAL_URI may be unavailable, if the user has deleted the downloaded file between when the download completed and the time you try to access the column.

There are a number of possible status codes (e.g., STATUS_FAILED, STATUS_SUCCESSFUL, STATUS_RUNNING). Some, like STATUS_FAILED, may have an accompanying reason to provide more details.

Note that you really should close this Cursor when you are done with it. StrictMode, for example, will complain if you do not.

Download Broadcasts

To find out about the results of the download, we need to register a BroadcastReceiver, to watch for two actions used by DownloadManager:

1. ACTION_DOWNLOAD_COMPLETE, to let us know when the download is done
2. ACTION_NOTIFICATION_CLICKED, to let us know if the user taps on the Notification displayed on the user's device related to our download

So, in onResume() of our fragment, we register a single BroadcastReceiver for both of those events:

```
@Override
public void onResume() {
  super.onResume();

  IntentFilter f=
      new IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLETE);

  f.addAction(DownloadManager.ACTION_NOTIFICATION_CLICKED);

  getActivity().registerReceiver(onEvent, f);
}
```

That BroadcastReceiver is unregistered in onPause():

```
@Override
public void onPause() {
  getActivity().unregisterReceiver(onEvent);

  super.onPause();
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `BroadcastReceiver` implementation examines the action string of the incoming `Intent` (via a call to `getAction()` and either displays a `Toast` (for `ACTION_NOTIFICATION_CLICKED`) or enables the start-download `Button`:

```
    public void onReceive(Context ctxt, Intent i) {
      if (DownloadManager.ACTION_NOTIFICATION_CLICKED.equals(i.getAction())) {
        Toast.makeText(ctxt, R.string.hi, Toast.LENGTH_LONG).show();
      }
      else {
        start.setEnabled(true);
      }
    }
  };
}
```

What the User Sees

The user, upon launching the application, sees our three pretty buttons:

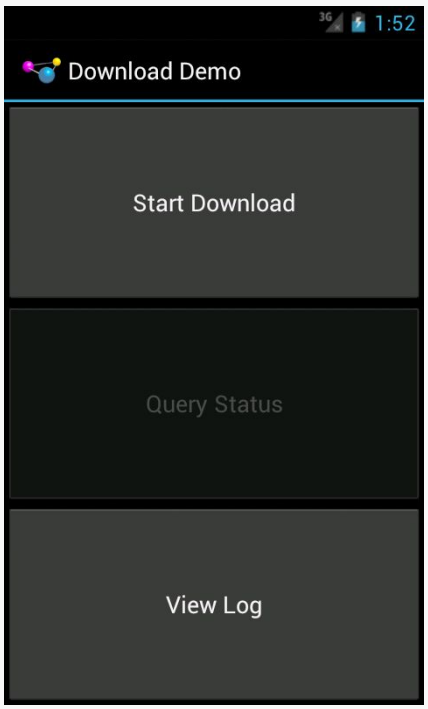


Figure 609: The Download Demo Sample, As Initially Launched

Clicking the first disables the button while the download is going on, and a download icon appears in the status bar (though it is a bit difficult to see, given the poor contrast between Android's icon and Android's status bar):

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

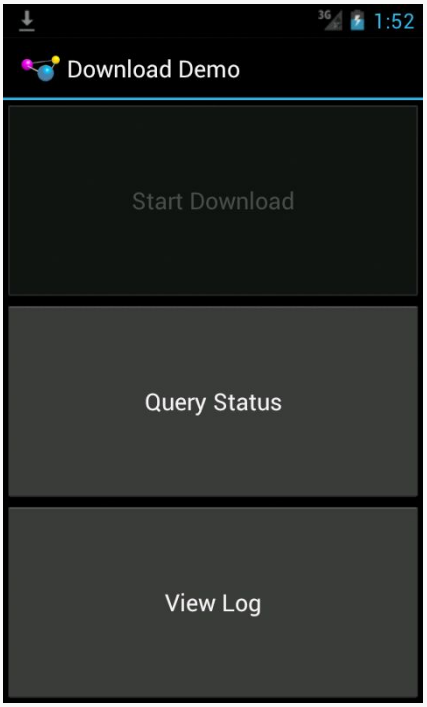


Figure 610: The Download Demo Sample, Downloading

Sliding down the notification drawer shows the user the progress in the form of a `ProgressBar` widget:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

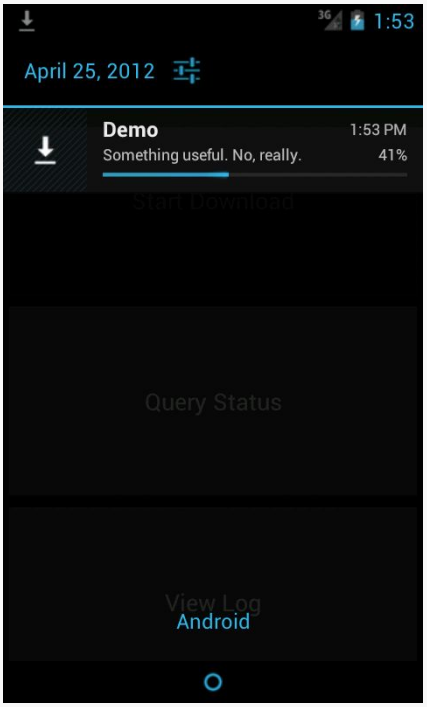


Figure 611: The DownloadManager Notification

Tapping on the entry in the notification drawer returns control to our original activity, where they see a `Toast`, raised by our `BroadcastReceiver`.

If they tap the middle button during the download, a different `Toast` will appear indicating that the download is in progress:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

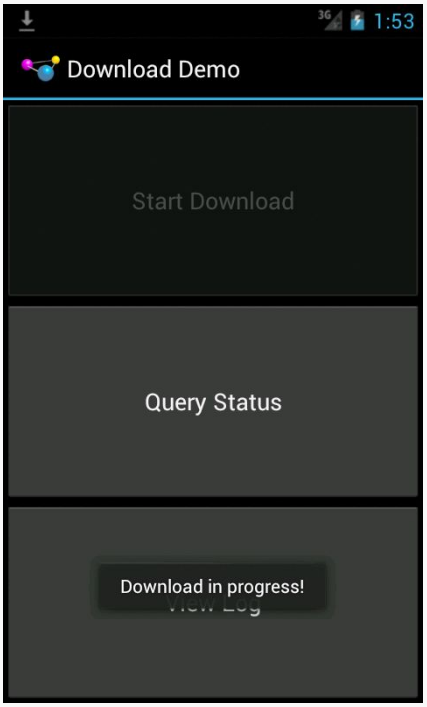


Figure 612: The Download Demo, Showing Download Status

Additional details are also dumped to LogCat, visible via DDMS or `adb logcat`:

```
12-10 08:45:01.289: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_ID: 12
12-10 08:45:01.289: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_BYTES_DOWNLOADED_SO_FAR: 615400
12-10 08:45:01.289: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_LAST_MODIFIED_TIMESTAMP: 1291988696232
12-10 08:45:01.289: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_LOCAL_URI: file:///mnt/sdcard/Download/test.mp4
12-10 08:45:01.299: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_STATUS: 2
12-10 08:45:01.299: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_REASON: 0
```

Once the download is complete, tapping the middle button will indicate that the download is, indeed, complete, and final information about the download is emitted to LogCat:

```
12-10 08:49:27.360: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_ID: 12
12-10 08:49:27.360: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_BYTES_DOWNLOADED_SO_FAR: 6219229
12-10 08:49:27.370: DEBUG/com.commonsware.android.download.DownloadDemo(372):
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
COLUMN_LAST_MODIFIED_TIMESTAMP: 1291988713409
12-10 08:49:27.370: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_LOCAL_URI: file:///mnt/sdcard/Download/test.mp4
12-10 08:49:27.370: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_STATUS: 8
12-10 08:49:27.370: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_REASON: 0
```

Tapping the bottom button brings up the activity displaying all downloads, including both successes and failures:

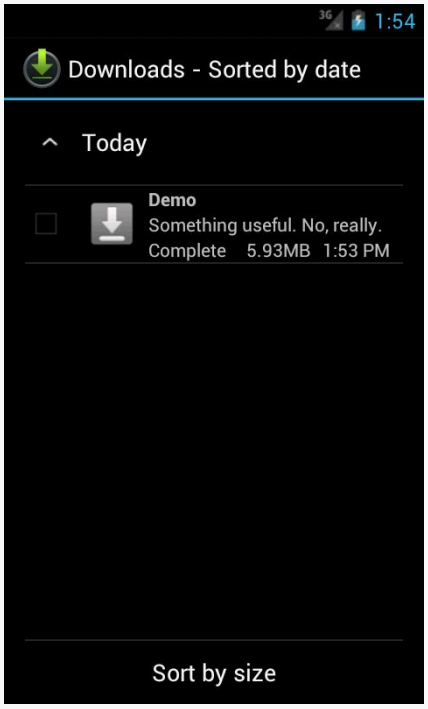


Figure 613: The DownloadManager Results

And, of course, the file is downloaded. In the emulator, our chosen location maps to `/mnt/sdcard/Downloads/test.mp4`.

Limitations

While `DownloadManager` nowadays supports HTTPS (SSL) URLs, that was not the case when it was introduced back in Android 2.3. You will want to test any HTTPS URLs you intend to use with `DownloadManager` if you are supporting older versions of Android.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you display the list of all downloads, and your download is among them, it is a really good idea to make sure that some activity (perhaps one of yours) is able to respond to an `ACTION_VIEW` `Intent` on that download's MIME type. Otherwise, when the user taps on the entry in the list, they will get a `Toast` indicating that there is nothing available to view the download. This may confuse users. Alternatively, use `setVisibleInDownloadsUi()` on your request, passing in `false`, to suppress it from this list.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Media

Special Creative Commons BY-NC-SA 4.0 License Edition

Audio Playback

Whether it comes in the form of simple beeps or in the form of symphonies (or gangster rap or whatever), Android applications often need to play audio. A few things in Android can play audio automatically, such as a `Notification`. However, once you get past those, you are on your own.

Fortunately for you, Android offers support for audio playback, and we will examine some of the options in this chapter.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Get Your Media On

In Android, you have five different places you can pull media clips from — one of these will hopefully fit your needs:

- You can package audio clips as raw resources (`res/raw` in your project), so they are bundled with your application. The benefit is that you're guaranteed the clips will be there; the downside is that they cannot be replaced without upgrading the application.
- You can package audio clips as assets (`assets/` in your project) and reference them via `file:///android_asset/` URLs in a `Uri`. The benefit over raw resources is that this location works with APIs that expect `Uri` parameters instead of resource IDs. The downside — assets are only replaceable when the application is upgraded — remains.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- You can store media in an application-local directory, such as content you download off the Internet. Your media may or may not be there, and your storage space isn't infinite, but you can replace the media as needed.
- You can store media — or make use of media that the user has stored herself — that is on an SD card. There is likely more storage space on the card than there is on the device, and you can replace the media as needed, but other applications have access to the SD card as well.
- You can, in some cases, stream media off the Internet, bypassing any local storage

Remember that on Android 1.x/2.x devices, internal storage space is at a premium. That means you should only package small clips in your app (`assets/` or `res/raw/`) and download larger clips to external storage.

MediaPlayer for Audio

If you want to play back music, particularly material in MP3 format, you will want to use the `MediaPlayer` class. With it, you can feed it an audio clip, start/stop/pause playback, and get notified on key events, such as when the clip is ready to be played or is done playing.

You have three ways to set up a `MediaPlayer` and tell it what audio clip to play:

- If the clip is a raw resource, use `MediaPlayer.create()` and provide the resource ID of the clip
- If you have a `Uri` to the clip, use the `Uri`-flavored version of `MediaPlayer.create()`
- If you have a string path to the clip, just create a `MediaPlayer` using the default constructor, then call `setDataSource()` with the path to the clip

Next, you need to call `prepare()` or `prepareAsync()`. Both will set up the clip to be ready to play, such as fetching the first few seconds off the file or stream. The `prepare()` method is synchronous; as soon as it returns, the clip is ready to play. The `prepareAsync()` method is asynchronous — more on how to use this version later.

Once the clip is prepared, `start()` begins playback, `pause()` pauses playback (with `start()` picking up playback where `pause()` paused), and `stop()` ends playback. One caveat: you cannot simply call `start()` again on the `MediaPlayer` once you have called `stop()` — we'll cover a workaround a bit later in this section.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To see this in action, take a look at the Media/Audio sample project. The layout is pretty trivial, with three buttons and labels for play, pause, and stop:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="4dip"
  >
    <ImageButton android:id="@+id/play"
      android:src="@drawable/play"
      android:layout_height="wrap_content"
      android:layout_width="wrap_content"
      android:paddingRight="4dip"
      android:enabled="false"
    />
    <TextView
      android:text="Play"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:gravity="center_vertical"
      android:layout_gravity="center_vertical"
      android:textAppearance="?android:attr/textAppearanceLarge"
    />
  </LinearLayout>
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="4dip"
  >
    <ImageButton android:id="@+id/pause"
      android:src="@drawable/pause"
      android:layout_height="wrap_content"
      android:layout_width="wrap_content"
      android:paddingRight="4dip"
    />
    <TextView
      android:text="Pause"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:gravity="center_vertical"
      android:layout_gravity="center_vertical"
      android:textAppearance="?android:attr/textAppearanceLarge"
    />
  </LinearLayout>
  <LinearLayout
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="4dip"
  >
    <ImageButton android:id="@+id/stop"
      android:src="@drawable/stop"
      android:layout_height="wrap_content"
      android:layout_width="wrap_content"
      android:paddingRight="4dip"
    />
    <TextView
      android:text="Stop"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:gravity="center_vertical"
      android:layout_gravity="center_vertical"
      android:textAppearance="?android:attr/textAppearanceLarge"
    />
  </LinearLayout>
</LinearLayout>
```

The Java, of course, is where things get interesting:

```
package com.commonsware.android.audio;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.Context;
import android.content.SharedPreferences;
import android.media.MediaPlayer;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.ImageButton;
import android.widget.Toast;

public class AudioDemo extends Activity
  implements MediaPlayer.OnCompletionListener {

  private ImageButton play;
  private ImageButton pause;
  private ImageButton stop;
  private MediaPlayer mp;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    play=(ImageButton)findViewById(R.id.play);
    pause=(ImageButton)findViewById(R.id.pause);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    stop=(ImageButton)findViewById(R.id.stop);

    play.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        play();
      }
    });

    pause.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        pause();
      }
    });

    stop.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        stop();
      }
    });

    setup();
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    if (stop.isEnabled()) {
      stop();
    }
  }

  public void onCompletion(MediaPlayer mp) {
    stop();
  }

  private void play() {
    mp.start();

    play.setEnabled(false);
    pause.setEnabled(true);
    stop.setEnabled(true);
  }

  private void stop() {
    mp.stop();
    pause.setEnabled(false);
    stop.setEnabled(false);

    try {
      mp.prepare();
      mp.seekTo(0);
      play.setEnabled(true);
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    catch (Throwable t) {
      goBlooey(t);
    }
  }

  private void pause() {
    mp.pause();

    play.setEnabled(true);
    pause.setEnabled(false);
    stop.setEnabled(true);
  }

  private void loadClip() {
    try {
      mp=MediaPlayer.create(this, R.raw.clip);
      mp.setOnCompletionListener(this);
    }
    catch (Throwable t) {
      goBlooey(t);
    }
  }

  private void setup() {
    loadClip();
    play.setEnabled(true);
    pause.setEnabled(false);
    stop.setEnabled(false);
  }

  private void goBlooey(Throwable t) {
    AlertDialog.Builder builder=new AlertDialog.Builder(this);

    builder
      .setTitle("Exception!")
      .setMessage(t.toString())
      .setPositiveButton("OK", null)
      .show();
  }
}
```

In `onCreate()`, we wire up the three buttons to appropriate callbacks, then call `setup()`. In `setup()`, we create our `MediaPlayer`, set to play a clip we package in the project as a raw resource. We also configure the activity itself as the completion listener, so we find out when the clip is over. Note that, since we use the static `create()` method on `MediaPlayer`, we have already implicitly called `prepare()`, so we do not need to call that separately ourselves.

The buttons simply work the `MediaPlayer` and toggle each others' states, via appropriately-named callbacks. So, `play()` starts `MediaPlayer` playback, `pause()` pauses playback, and `stop()` stops playback and resets our `MediaPlayer` to play

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

again. The `stop()` callback is also used for when the audio clip completes of its own accord.

To reset the `MediaPlayer`, the `stop()` callback calls `prepare()` on the existing `MediaPlayer` to enable it to be played again and `seekTo()` to move the playback point to the beginning. If we were using an external file as our media source, it would be better to call `prepareAsync()`.

The UI is nothing special, but we are more interested in the audio in this sample, anyway:

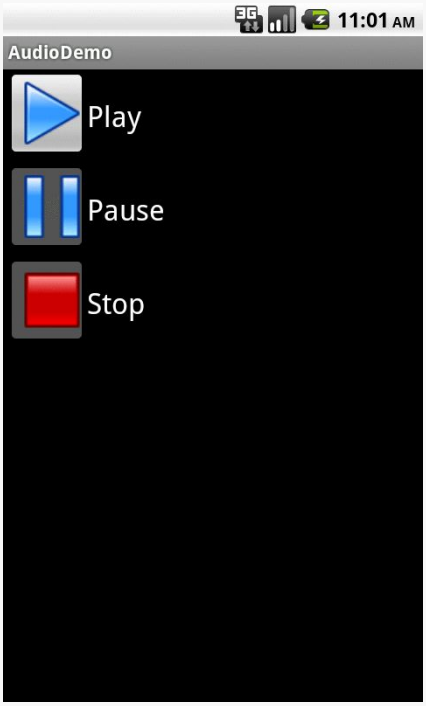


Figure 614: The AudioDemo sample application

Streaming Limitations

You can use the same basic code for streaming media, using an `http://` or `rtsp://` URL. However, bear in mind that Android does not support streaming MP3 over RTSP, as that exceeds the relevant RTSP specifications. That being said, there *are* MP3-over-RTSP streams in the world, and clients and servers that have negotiated an ad-hoc extension to the specification to accommodate this. Android cannot play these streams.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Other Ways to Make Noise

While `MediaPlayer` is the primary audio playback option, particularly for content along the lines of MP3 files, there are other alternatives if you are looking to build other sorts of applications, notably games and custom forms of streaming audio.

SoundPool

The `SoundPool` class's claim to fame is the ability to overlay multiple sounds, and do so in a prioritized fashion, so your application can just ask for sounds to be played and `SoundPool` deals with each sound starting, stopping, and blending while playing.

This may make more sense with an example.

Suppose you are creating a first-person shooter. Such a game may have several sounds going on at any one time:

1. The sound of the wind whistling amongst the trees on the battlefield
2. The sound of the surf crashing against the beach in the landing zone
3. The sound of booted feet crunching on the sand
4. The sound of the character's own panting as the character runs on the beach
5. The sound of orders being barked by a sergeant positioned behind the character
6. The sound of machine gun fire aimed at the character and the character's squad mates
7. The sound of explosions from the gun batteries of the battleship providing suppression fire

And so on.

In principle, `SoundPool` can blend all of those together into a single audio stream for output. Your game might set up the wind and surf as constant background sounds, toggle the feet and panting on and off based on the character's movement, randomly add the barked orders, and tie the gunfire based on actual game play.

In reality, your average smartphone will lack the CPU power to handle all of that audio without harming the frame rate of the game. So, to keep the frame rate up, you tell `SoundPool` to play at most two streams at once. This means that when nothing else is happening in the game, you will hear the wind and surf, but during

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the actual battle, those sounds get dropped out — the user might never even miss them — so the game speed remains good.

AudioTrack

The lowest-level Java API for playing back audio is `AudioTrack`. It has two main roles:

1. Its primary role is to support streaming audio, where the streams come in some format other than what `MediaPlayer` handles. While `MediaPlayer` can handle RTSP, for example, it does not offer SIP. If you want to create a SIP client (perhaps for a VOIP or Web conferencing application), you will need to convert the incoming data stream to PCM format, then hand the stream off to an `AudioTrack` instance for playback.
2. It can also be used for "static" (versus streamed) bits of sound that you have pre-decoded to PCM format and want to play back with as little latency as possible. For example, you might use this for a game for in-game sounds (beeps, bullets, or "boing"s). By pre-decoding the data to PCM and caching that result, then using `AudioTrack` for playback, you will use the least amount of overhead, minimizing CPU impact on game play and on battery life.

ToneGenerator

If you want your phone to sound like... well... a phone, you can use `ToneGenerator` to have it play back dual-tone multi-frequency (DTMF) tones. In other words, you can simulate the sounds played by a regular "touch-tone" phone in response to button presses. This is used by the Android dialer, for example, to play back the tones when users dial the phone using the on-screen keypad, as an audio reinforcement.

Note that these will play through the phone's earpiece, speaker, or attached headset. They do not play through the outbound call stream. In principle, you might be able to get `ToneGenerator` to play tones through the speaker loud enough to be picked up by the microphone, but this probably is not a recommended practice.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Audio Recording

Most Android devices have microphones. On such devices, it might be nice to get audio input from those microphones, whether to record locally, process locally (e.g., speech recognition), or to stream out over the Internet (e.g., voice over IP).

Not surprisingly, Android has some capabilities in this area. Also, not surprisingly, there are multiple APIs, with varying mixes of power and complexity, to allow you to capture microphone input. In this chapter, we will examine `MediaRecorder` for recording audio files and `AudioRecord` for raw microphone input.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Having read the chapter on audio playback is probably also a good idea. And, for the section on playing back local streams, you will want to have read up on content providers, particularly the chapter on provider patterns.

Recording by Intent

Just as the easiest way to take a picture with the camera is to use the device's built-in camera app, the easiest way to record some audio is to use a built-in activity for it. And, as with using the built-in camera app, the built-in audio recording activity has some significant limitations.

Requesting the built-in audio recording activity is a matter of calling `startActivityForResult()` for a `MediaStore.Audio.Media.RECORD_SOUND_ACTION` action. You can see this in the `Media/SoundRecordIntent` sample project, specifically the `MainActivity`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.soundrecord;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.provider.MediaStore;
import android.widget.Toast;

public class MainActivity extends Activity {
  private static final int REQUEST_ID=1337;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Intent i=new Intent(MediaStore.Audio.Media.RECORD_SOUND_ACTION);

    startActivityForResult(i, REQUEST_ID);
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode == REQUEST_ID && resultCode == RESULT_OK) {
      Toast.makeText(this, "Recording finished!", Toast.LENGTH_LONG)
           .show();
    }

    finish();
  }
}
```

As with a few other sample apps in this book, the `Media/SoundRecordIntent` uses a `Theme.NoDisplay` activity, eschewing its own UI. Instead, in `onCreate()`, we immediately call `startActivityForResult()` for `MediaStore.Audio.Media.RECORD_SOUND_ACTION`. That will bring up a recording activity:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

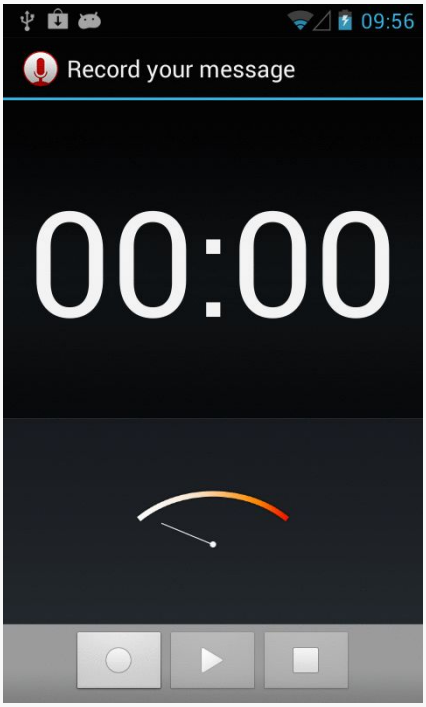


Figure 615: Built-In Sound Recording Activity

If the user records some audio via the "record" `ImageButton` (one with the circle icon) and the "stop" `ImageButton` (one with the square icon), you will get control back in `onActivityResult()`, where you are passed an `Intent` whose `Uri` (via `getData()`) will point to this audio recording in the `MediaStore`.

However:

- You have no control over where the file is stored or what it is named. It appears that, by default, these files are dumped unceremoniously in the root of external storage.
- You have no control over anything about the way the audio is recorded, such as codecs or bitrates. For example, it appears that, by default, the files are recorded in AMR format.
- `ACTION_VIEW` may not be able to play back this audio (leastways, it failed to in testing on a few devices). Whether that is due to codecs, the way the data is put in `MediaStore`, or the limits of the default audio player on Android, is unclear.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Hence, in many cases, while this works, it may not work well enough — or controlled enough — to meet your needs. In that case, you will want to handle the recording yourself, as will be described in the next couple of sections.

Recording to Files

If your objective is to record a voice note, a presentation, or something along those lines, then `MediaRecorder` is probably the class that you want. It will let you specify what sort of media you wish to record, in what format, and to what location. It then handles the actual act of recording.

To illustrate this, let us review the Media/AudioRecording sample project.

Our activity's layout consists of a single `ToggleButton` widget named `record`:

```
<ToggleButton xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/record"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:textAppearance="?android:attr/textAppearanceLarge"/>
```

In `onCreate()` of `MainActivity`, we load the layout and set the activity itself up as the `OnCheckedChangedListener`, to find out when the user toggles the button:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    ((ToggleButton)findViewById(R.id.record)).setOnCheckedChangeListener(this);
  }
```

Also, in `onResume()`, we initialize a `MediaRecorder`, setting the activity up as being the one to handle info and error events about the recording. Similarly, we `release()` the `MediaRecorder` in `onPause()`, to reduce our overhead when we are not in the foreground:

```
  @Override
  public void onResume() {
    super.onResume();

    recorder=new MediaRecorder();
    recorder.setOnErrorListener(this);
    recorder.setOnInfoListener(this);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void onPause() {
    recorder.release();
    recorder=null;

    super.onPause();
  }
```

Most of the work occurs in `onCheckedChanged()`, where we get control when the user toggles the button. If we are now checked, we begin recording; if not, we stop the previous recording:

```
  @Override
  public void onCheckedChanged(CompoundButton buttonView,
                               boolean isChecked) {
    if (isChecked) {
      File output=
          new File(

Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS),
                   BASENAME);

      recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
      recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
      recorder.setOutputFile(output.getAbsolutePath());

      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD_MR1) {
        recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
        recorder.setAudioEncodingBitRate(160 * 1024);
      }
      else {
        recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
      }

      recorder.setAudioChannels(2);

      try {
        recorder.prepare();
        recorder.start();
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(),
              "Exception in preparing recorder", e);
        Toast.makeText(this, e.getMessage(), Toast.LENGTH_LONG).show();
      }
    }
    else {
      try {
        recorder.stop();
      }
      catch (Exception e) {
        Log.w(getClass().getSimpleName(),
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
            "Exception in stopping recorder", e);
        // can fail if start() failed for some reason
      }

      recorder.reset();
    }
  }
```

To record audio, we:

- Create a `File` object representing where the recording should be stored, in this case using `Environment.getExternalStoragePublicDirectory()` to find a location on external storage
- Tell the `MediaRecorder` that we wish to record from the microphone, through a call to `setAudioSource()`, that we wish to record a 3GP file via a call to `setOutputFormat()`, and that we wish to record the results to our `File` via a call to `setOutputFile()`
- If we are running on Android 2.3.3 or higher, we can also configure our encoder to be AAC via `setAudioEncoder()` and set our requested bitrate to 160Kbps via `setAudioEncodingBitRate()` — otherwise, we use `setAudioEncoder()` to request AMR narrowband
- Indicate how many audio channels we want via `setAudioChannels()`, such as `2` to attempt to record in stereo
- Kick off the actual recording via calls to `prepare()` (to set up the output file) and `record()`

Stopping the recording, when the user toggles off the button, is merely a matter of calling `stop()` on the `MediaRecorder`.

Because we told the `MediaRecorder` that our activity was our `OnErrorListener` and `OnInfoListener`, we have to implement those interfaces on the activity and implement their required methods (`onError()` and `onInfo()`, respectively). In the normal course of events, neither of these should be triggered. If they are, we are passed an `int` value (typically named `what`) that indicates what happened:

```
  @Override
  public void onInfo(MediaRecorder mr, int what, int extra) {
    String msg=getString(R.string.strange);

    switch (what) {
      case MediaRecorder.MEDIA_RECORDER_INFO_MAX_DURATION_REACHED:
        msg=getString(R.string.max_duration);
        break;

      case MediaRecorder.MEDIA_RECORDER_INFO_MAX_FILESIZE_REACHED:
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      msg=getString(R.string.max_size);
      break;
    }

    Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
  }

  @Override
  public void onError(MediaRecorder mr, int what, int extra) {
    Toast.makeText(this, R.string.strange, Toast.LENGTH_LONG).show();
  }
```

Here, we just raise a `Toast` in either case, with either a generic message or a specific message for the cases where the maximum time duration or the maximum file size for our recording has been reached.

We also need to hold the `RECORD_AUDIO` and `WRITE_EXTERNAL_STORAGE` permissions. `RECORD_AUDIO`, in particular, is needed to let the user know that we intend to record information off of the microphone.

The results are that we get a recording on external storage (typically in a `Downloads` directory) after we toggle the button on, record some audio, then toggle the button off.

`MediaRecorder` is rather fussy about the order of method calls for its configuration. For example, you must call `setAudioEncoder()` *after* the call to `setOutputFormat()`.

Also, the available codecs and file types are rather limited. Notably, Android lacks the ability to record to MP3 format, perhaps due to patent licensing issues.

On the flip side, `MediaRecorder` also supports recording video, a topic which is not presently covered in this book.

Recording to Streams

The nice thing about recording to files is that Android handles all of the actual file I/O for us. The downside is that because Android handles all of the actual file I/O for us, it can only write files that are accessible to it and our process, meaning external storage. This may not be suitable in all cases, such as wanting to record to some form of private encrypted storage.

The good news is that Android does support recording to streams, in the form of a pipe created by `ParcelFileDescriptor` and `createPipe()`. This follows the same

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

basic pattern that we saw in the chapter on content provider patterns, where we served a stream via a pipe. However, as you will see, there are some limits on how well we can do this.

To demonstrate and explain, let us examine the Media/AudioRecordStream sample project. This is nearly a complete clone of the previous sample, so we will only focus on the changes in this section.

The author would like to thank Lucio Maciel for his assistance in getting this example to work.

Setting Up the Stream

The biggest change, by far, is in our `setOutputFile()` call. Before, we supplied a path to external storage. Now, we supply the write end of a pipe:

```
      recorder.setOutputFile(getStreamFd());
```

Our `getStreamFd()` method looks a lot like the `openFile()` method of our pipe-providing provider:

```
  private FileDescriptor getStreamFd() {
    ParcelFileDescriptor[] pipe=null;

    try {
      pipe=ParcelFileDescriptor.createPipe();

      new TransferThread(new AutoCloseInputStream(pipe[0]),
                         new FileOutputStream(getOutputFile())).start();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception opening pipe", e);
    }

    return(pipe[1].getFileDescriptor());
  }
```

We create our pipe with `createPipe()`, spawn a `TransferThread` to copy the recording from an `InputStream` to a `FileOutputStream`, and return the write end of the pipe. However, `setOutputFile()` on `MediaRecorder` takes the actual integer file descriptor, not a `ParcelFileDescriptor`, so we use `getFileDescriptor()` to retrieve the file descriptor and return that.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Our `TransferThread` is similar to the one from the content provider sample, except that we pass over a `FileOutputStream`, so we can not only `flush()` but also `sync()` when we are done writing:

```
static class TransferThread extends Thread {
  InputStream in;
  FileOutputStream out;

  TransferThread(InputStream in, FileOutputStream out) {
    this.in=in;
    this.out=out;
  }

  @Override
  public void run() {
    byte[] buf=new byte[8192];
    int len;

    try {
      while ((len=in.read(buf)) >= 0) {
        out.write(buf, 0, len);
      }

      in.close();

      out.flush();
      out.getFD().sync();
      out.close();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(),
            "Exception transferring file", e);
    }
  }
}
```

Changes in Recording Configuration

The biggest limitation of a pipe's stream is that it is *purely* a stream. You cannot rewind re-read earlier bits of data. In other words, the stream is not seekable.

That is a problem with `MediaRecorder` in some configurations. For example, a 3GP file contains a header with information about the overall file, information that `MediaRecorder` does not know until the recording is complete. In the case of a file, `MediaRecorder` can simply rewind and update the header with the final data when everything is done. However, that is not possible with a pipe-based stream.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, some configurations will work, notably "raw" ones that just have the recorded audio, with no type of header. That is what we use in this sample.

Specifically, we now write to a `.amr` file:

```
  private static final String BASENAME="recording-stream.amr";
```

We also set our output format to `RAW_AMR`, and our encoder to `AMR_NB`:

```
      recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
      recorder.setOutputFormat(MediaRecorder.OutputFormat.RAW_AMR);
      recorder.setOutputFile(getStreamFd());
      recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
      recorder.setAudioChannels(2);
```

This combination works. Other combinations might also work. But our approach of writing the 3GP file, as in the file-based example, will not work.

Raw Audio Input

Just as `AudioTrack` allows you to play audio supplied as raw 8- or 16-bit PCM input, `AudioRecord` allows you to record audio from the microphone, supplied to you in PCM format. It is then up to you to actually do something with the raw byte PCM data, including converting it to some other format and container as needed.

Note that you need `RECORD_AUDIO` to work with `AudioRecord`, just as you need it to work with `MediaRecorder`.

Requesting the Microphone

As noted in the opening paragraph of this chapter, most Android devices have microphones. The key word there is *most*. Not all Android devices will have microphones, as only some tablets (and fewer Android TV devices) will support microphone input.

As with most of this optional hardware, the solution is to use `<uses-feature>`. In that case, you would request the `android.hardware.microphone` feature, with `android:required="false"` if you felt that you do not absolutely *need* a microphone. In that case, you would use `hasSystemFeature()` on `PackageManager` to determine at runtime if you do indeed have a microphone.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that the `RECORD_AUDIO` permission implies that you need a microphone. Hence, even if you skip the `<uses-feature>` element, your app will still only ship to devices that have a microphone. If the microphone is optional, be sure to include `android:required="false"`, so your app will be available to devices that lack a microphone.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Video Playback

Just as Android supports audio playback, it also supports video playback of local and streaming content. Unlike audio playback – which supports a mix of high-level and low-level APIs – video playback offers a purely high-level interface, in the form of the same `MediaPlayer` class you used for audio playback. To keep things a bit simpler, though, Android does offer a `VideoView` widget you can drop in an activity or fragment to play back video.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, along with the chapter on audio playback.

Moving Pictures

Video clips get their own widget, the `VideoView`. Put it in a layout, feed it an MP4 video clip, and you get playback!

For example, take a look at this layout, from the `Media/Video` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <VideoView
     android:id="@+id/video"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    />
</LinearLayout>
```

The layout is simply a full-screen video player. Whether it will use the full screen will be dependent on the video clip, its aspect ratio, and whether you have the device (or emulator) in portrait or landscape mode.

Wiring up the Java is almost as simple:

```
package com.commonsware.android.video;

import java.io.File;
import android.app.Activity;
import android.graphics.PixelFormat;
import android.os.Bundle;
import android.os.Environment;
import android.widget.MediaController;
import android.widget.VideoView;

public class VideoDemo extends Activity {
  private VideoView video;
  private MediaController ctlr;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    getWindow().setFormat(PixelFormat.TRANSLUCENT);
    setContentView(R.layout.main);

    File clip=new File(Environment.getExternalStorageDirectory(),
                       "test.mp4");

    if (clip.exists()) {
      video=(VideoView)findViewById(R.id.video);
      video.setVideoPath(clip.getAbsolutePath());

      ctlr=new MediaController(this);
      ctlr.setMediaPlayer(video);
      video.setMediaController(ctlr);
      video.requestFocus();
      video.start();
    }
  }
}
```

Here, we:

1. Confirm that our video file exists on external storage
2. Tell the `VideoView` which file to play

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

3. Create a `MediaController` pop-up panel and cross-connect it to the `VideoView`
4. Give the `VideoView` the focus and start playback

The biggest trick with `VideoView` is getting a video clip onto the device. While `VideoView` does support some streaming video, the requirements on the MP4 file are fairly stringent. If you want to be able to play a wider array of video clips, you need to have them on the device, preferably on an SD card.

The crude `VideoDemo` class assumes there is an MP4 file named `test.mp4` in the root of external storage on your device or emulator. Once there, the Java code shown above will give you a working video player:

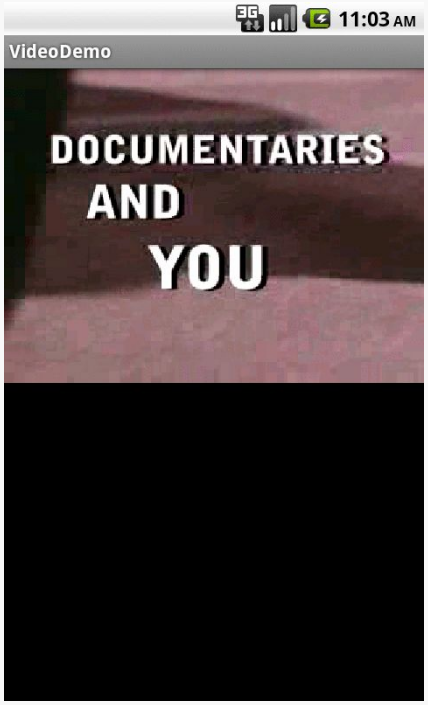


Figure 616: The VideoDemo sample application, showing a Creative Commons-licensed video clip

Tapping on the video will pop up the playback controls:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 617: The VideoDemo sample application, with the media controls displayed

The video will scale based on space, as shown in this rotated view of the emulator (`<Ctrl>-<F12>`):

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

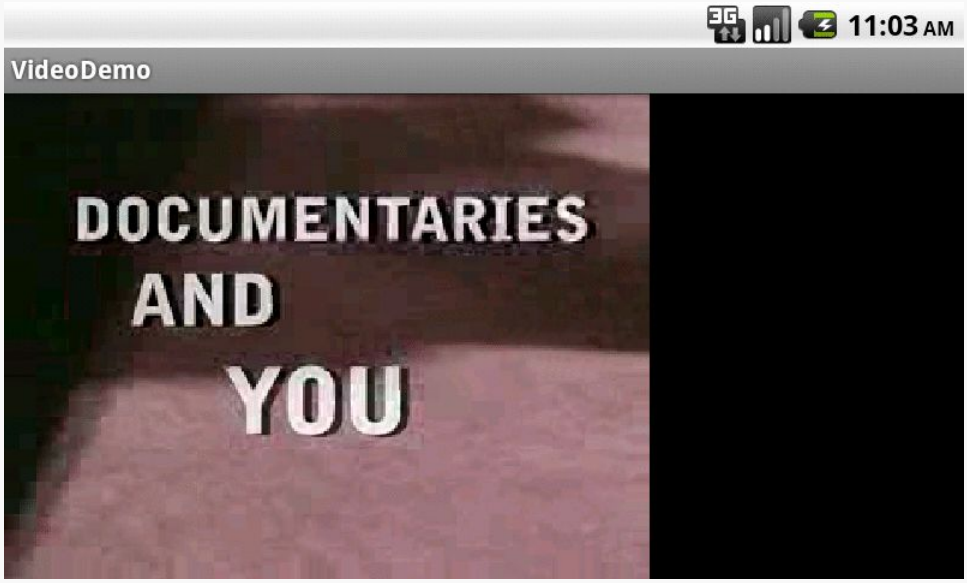


Figure 618: The VideoDemo sample application, in landscape mode, with the video clip scaled to fit

NOTE: playing video on the Android emulator may work for you, but it is not terribly likely. Video playback requires graphic acceleration to work well, and the emulator does not have graphics acceleration — regardless of the capabilities of the actual machine the emulator runs on. Hence, if you try playing back video in the emulator, expect problems. If you are serious about doing Android development with video playback, you definitely need to acquire a piece of Android hardware.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Using the Camera via 3rd-Party Apps

Most Android devices will have a camera, since they are fairly commonplace on mobile devices these days. You, as an Android developer, can take advantage of the camera, for everything from snapping tourist photos to scanning barcodes. If you wish to let other apps do the "heavy lifting" for you, working with the camera can be fairly straightforward. If you want more control, you can work with the camera directly, though this control comes with greater complexity.

You can also record videos using the camera. Once again, you have the option of either using a third-party activity, or doing it yourself.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the material on implicit `Intents`.

Being Specific About Features

If your app needs a camera — by any of the means cited in this chapter – you should include a `<uses-feature>` element in the manifest indicating your requirements. However, you need to be fairly specific about your requirements here.

For example, the Nexus 7 has a camera... but only a front-facing camera. This facilitates apps like video chat. However, the `android.hardware.camera` implies that you need a high-resolution rear-facing camera, even though this is undocumented. Hence, to work with the Nexus 7's camera, you need to:

- Require the `CAMERA` permission (if you are using the `Camera` directly)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Not require the `android.hardware.camera` feature (`android:required="false"`)
- Optionally require the `android.hardware.camera.front` feature (if your app definitely needs a front-facing camera)

At runtime, you would use `hasSystemFeature()` on `PackageManager`, or interrogate the `Camera` class for available cameras, to determine what you have access to.

Note that if you want to record audio when recording videos, you should also consider the `android.hardware.microphone` feature.

Still Photos: Letting the Camera App Do It

The easiest way to take a picture is to not take the picture yourself, but let somebody else do it. The most common implementation of this approach is to use an `ACTION_IMAGE_CAPTURE Intent` to bring up the user's default camera application, and let it take a picture on your behalf.

To see this in use, take a look at the `Camera/Content` sample project. This trivial app will use system-supplied activities to take a picture, then view the result, without actually implementing any of its own UI.

The Implementation

Of course, we still need an activity, so our code can be launched by the user. We just set it up with `Theme.NoDisplay`, so no UI will be created for it:

```
    <activity
      android:name=".CameraContentDemoActivity"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
```

The activity itself — `CameraContentDemoActivity` — consists solely of `onCreate()` and `onActivityResult()` methods:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onCreate(savedInstanceState);

    Intent i=new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    File dir=

Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DCIM);

    output=new File(dir, "CameraContentDemo.jpeg");
    i.putExtra(MediaStore.EXTRA_OUTPUT, Uri.fromFile(output));

    startActivityForResult(i, CONTENT_REQUEST);
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode == CONTENT_REQUEST) {
      if (resultCode == RESULT_OK) {
        Intent i=new Intent(Intent.ACTION_VIEW);

        i.setDataAndType(Uri.fromFile(output), "image/jpeg");
        startActivity(i);
        finish();
      }
    }
  }
}
```

In `onCreate()`, we create our `ACTION_IMAGE_CAPTURE` `Intent`. We add an extra, keyed as `MediaStore.EXTRA_OUTPUT`, indicating where we want the app to save the resulting picture. In our case, we store that in a `CameraContentDemo.jpeg` file in the default external storage directory for photos (identified by `Environment.DIRECTORY_DCIM`). The documentation for `ACTION_IMAGE_CAPTURE` indicates that this needs to be in the form of a `Uri` object, which is why we use `Uri.fromFile()` to convert our string path into the `Uri`.

At that point, we call `startActivityForResult()` to bring up the user's chosen camera app to take our picture. We next get control in `onActivityResult()`. There, we create an `ACTION_VIEW` `Intent`, pointing at our output file, indicating the MIME type is `image/jpeg`, and start up an activity for that. This should bring up the Gallery or another app capable of displaying the photo on the screen.

The Caveats

There are several downsides to this approach.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

First, you have no control over the camera app itself. You do not even really know what app it is. You cannot dictate certain features that you would like (e.g., resolution, color effects). You simply blindly ask for a photo and get the result.

Also, since you do not know what the camera app is or behaves like, you cannot document that portion of your application's flow very well. You can say things like "at this point, you can take a picture using your chosen camera app", but that is about as specific as you can get.

Finally, some camera apps misbehave, returning odd results, such as a thumbnail-sized image rather than a max-resolution image. There is little you can do about this.

So, while this approach is easy, it may pose some quality-control issues.

Scanning with ZXing

If your objective is to scan a barcode, it is *much* simpler for you to integrate Barcode Scanner into your app than to roll it yourself.

Barcode Scanner – one of the most popular Android apps of all time — can scan a wide range of 1D and 2D barcode types. They offer an integration library that you can add to your app to initiate a scan and get the results. The library will even lead the user to the Play Store to install Barcode Scanner if they do not already have the app.

One limitation is that while the ZXing team (the authors and maintainers of Barcode Scanner) make the integration library available, they only do so in source form, requiring you to check out a bunch of source code and run a command-line build to get a JAR. Or, you can download a JAR that is used in the sample project for this section, if you prefer.

That sample project — `Camera/ZXing` – has a UI dominated by a "Scan!" button. Clicking the button invokes a `doScan()` method in our sample activity:

```
public void doScan(View v) {
  (new IntentIntegrator(this)).initiateScan();
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This passes control to Barcode Scanner by means of the integration JAR and the `IntentIntegrator` class. `initiateScan()` will validate that Barcode Scanner is installed, then will start up the camera and scan for a barcode.

Once Barcode Scanner detects a barcode and decodes it, the activity invoked by `initiateScan()` finishes, and control returns to you in `onActivityResult()` (as the Barcode Scanner scanning activity was invoked via `startActivityForResult()`). There, you can once again use `IntentIntegrator` to find out details of the scan, notably the type of barcode and the encoded contents:

```
public void onActivityResult(int request, int result, Intent i) {
  IntentResult scan=IntentIntegrator.parseActivityResult(request,
                                                         result,
                                                         i);

  if (scan!=null) {
    format.setText(scan.getFormatName());
    contents.setText(scan.getContents());
  }
}
```

Some notes:

- Barcode Scanner's scanning activity only works in landscape
- Even though you are not using the camera directly yourself, you should consider including the `<uses-feature>` element declaring that you need a camera, if your app cannot function without barcodes
- If you wish to add Barcode Scanner logic directly to your app, and avoid the dependency on the third-party APK, that is possible, but the process for doing it is not well documented

Videos: Letting the Camera App Do It

Just as `ACTION_IMAGE_CAPTURE` can be used to have a third-party app supply you with still images, there is an `ACTION_VIDEO_CAPTURE` on `MediaStore` that can be used as an `Intent` action for asking a third-party app capture a video for you. As with `ACTION_IMAGE_CAPTURE`, you use `startActivityForResult()` with `ACTION_VIDEO_CAPTURE` to find out when the video has been recorded.

There are two extras of note for `ACTION_VIDEO_CAPTURE`:

- `MediaStore.EXTRA_OUTPUT`, which indicates where on external storage the video should be written, and

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `MediaStore.EXTRA_VIDEO_QUALITY`, which should be an integer, either `0` for low quality/low size videos or `1` for high quality

If you elect to skip `EXTRA_OUTPUT`, the video will be written to the default directory for videos on the device (typically a "Movies" directory in the root of external storage), and the `Uri` you receive on the `Intent` in `onActivityResult()` will point to this file.

The impacts of skipping `EXTRA_VIDEO_QUALITY` are undocumented.

The Media/VideoRecordIntent sample project is a near-clone of the `Camera/Content` sample from earlier in this chapter. Instead of requesting a third-party app take a still image, though, this sample requests that a third-party app record a video:

```
package com.commonsware.android.videorecord;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.os.Environment;
import android.provider.MediaStore;
import java.io.File;

public class MainActivity extends Activity {
  private static final int REQUEST_ID=1337;
  private Uri result=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    File video=
        new File(

Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_MOVIES),
                 "sample.mp4");

    if (video.exists()) {
      video.delete();
    }

    Intent i=new Intent(MediaStore.ACTION_VIDEO_CAPTURE);

    result=Uri.fromFile(video);
    i.putExtra(MediaStore.EXTRA_OUTPUT, result);
    i.putExtra(MediaStore.EXTRA_VIDEO_QUALITY, 1);

    startActivityForResult(i, REQUEST_ID);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode == REQUEST_ID && resultCode == RESULT_OK) {
      Intent view=
          new Intent(Intent.ACTION_VIEW).setDataAndType(result,
                                                        "video/mp4");

      startActivity(view);
    }

    finish();
  }
}
```

`onCreate()` of `MainActivity` starts by setting up a `File` object pointing to a `sample.mp4` file in the standard location for movies on the device. If the file already exists, `onCreate()` deletes it. Then, `onCreate()` sets up the `Intent` to request that a third-party app record a movie to that location, at high quality (`EXTRA_VIDEO_QUALITY` set to `1`).

The call to `startActivityForResult()` will trigger the third-party app to record the video. When control returns to `MainActivity`, `onActivityResult()` creates an `ACTION_VIEW` `Intent` for the same file, then calls `startActivity()` to request that some app play back the video.

Directly Working with the Camera

Of course, you can bypass these third-party apps and work directly with the camera if you so choose. This is very painful... unless you use some wrapper code published by the author of this book, in which case working with the camera directly is somewhat less painful than normal.

The next chapter will explain all of this in great detail.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Working Directly with the Camera

Letting third-party apps take the pictures and videos for you is all well and good, but there will be times where you need more control than that. It is possible for you to work directly with the device cameras, via the `Camera` class.

(note that there are two classes in Android named `Camera` — the one of relevance for this chapter is `android.hardware.Camera`)

Working with `Camera` is ridiculously complicated, which is why the author of this book has created a `CameraFragment` to help streamline the implementation of a camera-using app.

Hence, first, we will cover the use of `CameraFragment` for simple picture-taking scenarios — this may be all that you need for your particular app.

As we get into more complex use cases for `CameraFragment`, we will explain a bit about the `Camera` and `MediaRecorder` APIs that `CameraFragment` uses "under the covers", so you have some context for how to work with `CameraFragment` and some idea of what you need to do if `CameraFragment` is ill-suited to your particular use case.

NOTE: The `Camera` API covered here — `android.hardware.Camera` — is deprecated as of Android 5.0. However, it is highly unlikely that there will ever be a backport of the replacement `android.hardware.camera2` API. Hence, you will need to use the `Camera` API until such time as your `minSdkVersion` rises to 21 or higher. The `CameraFragment` discussed in this chapter also uses the `Camera` API, not the new `android.hardware.camera2` API.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Prerequisites

This chapter assumes that you have read the previous chapter covering `Intent`-based uses of the camera and the chapter on audio recording.

Basic CameraFragment Usage

`CameraFragment` resides in the CWAC-Camera project. While in principle it is an Android library project, in practice you will probably use the published JAR instead, or integrate it as an AAR for your Gradle-based build.

With that in mind the steps for adding a `CameraFragment` to your app include:

Step #1: Download the JAR and put it in the `libs/` directory of your project (or, if you prefer, clone the GitHub repo and add it as a library project to your main project). Alternatively, follow the instructions on the CWAC-Camera project to add it to your `build.gradle` file.

Step #2: Add a `CameraFragment` to your UI. You have two versions of `CameraFragment` to choose from:

- `com.commonsware.cwac.camera.CameraFragment` for use with native API Level 11+ fragments
- `com.commonsware.cwac.camera.acl.CameraFragment` for use with the Android Support package's backport of fragments and ActionBarSherlock, supporting API Level 9 and 10

(note: if you choose the latter, your project will also need to have the ActionBarSherlock library project, and you will need a second CWAC-Camera JAR file containing these backported classes)

The `CameraFragment` is responsible for rendering your preview, so you need to size and position it as desired.

Step #3: Call `takePicture()` on the `CameraFragment` when you want to take a picture, which will be stored in the default digital photos directory (e.g., `DCIM`) on external storage as `Photo_yyyyMMdd_HHmmss.jpg`, where `yyyyMMdd_HHmmss` is replaced by the current date and time.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Step #3b: Call `startRecording()` and `stopRecording()` on the `CameraFragment` to record a video. **NOTE** that this is presently only available on `com.commonsware.cwac.camera.CameraFragment` for use with native API Level 11+ fragments. The resulting video will be stored in the default videos directory (e.g., `Movies`) on external storage as `Video_yyyyMMdd_HHmmss.mp4`, where `yyyyMMdd_HHmmss` is replaced by the current date and time.

Step #4: Optionally add `android:largeHeap="true"` to the `<application>` element in the manifest.

And that's it.

`CameraFragment` (and its underlying `CameraView`) will handle:

- Showing the preview using an optimal preview frame size, and managing the aspect ratio of the on-screen preview `View` so that your previews do not appear stretched
- Dealing with configuration changes and screen rotation, so your camera activity can work in portrait or landscape
- Following the appropriate recipes for taking still pictures and videos, including choosing the largest-available image size for the resolution
- Opening and closing the camera at the appropriate times, so when you are in the foreground you have exclusive camera access, but other apps will have access to the camera while your activity is not in the foreground
- And more!

Simple Configuration and Usage

Of course, there are probably plenty of things that you will want to configure about the process of taking photos and videos. There are many hooks in `CWAC-Camera` to allow you to do just that.

Much of this configuration involves creating a custom `CameraHost`. `CameraHost` is your primary interface with the `CWAC-Camera` classes for configuring the behavior of the camera. `CameraHost` is an interface, one that you are welcome to implement in full. Most times, though, you will be better served extending `SimpleCameraHost`, the default implementation of `CameraHost`, so that you can override only those methods where you want behavior different from the default.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`SimpleCameraHost` also offers `SimpleCameraHost.Builder` that you can use for some simple configuration, instead of creating a subclass of `SimpleCameraHost` and overriding methods. Create an instance of the `Builder` (passing in a `Context` to the constructor, such as your activity), call the various builder-style setters for configuration, and call `build()` to get your customized `SimpleCameraHost` instance.

You can pass your customized instance of `CameraHost` to `setHost()` on your `CameraFragment`, to replace the default. **Do this in `onCreate()` of a `CameraFragment` subclass** (or, if practical, just after instantiating your fragment) to ensure that the right `CameraHost` is used everywhere.

PictureTransaction

The `takePicture()` method has a zero-argument version that just takes a picture. It also has a one-argument version, where the argument is an instance of `PictureTransaction`. This will allow you to configure details of this particular picture to be taken. You can create a `PictureTransaction` using its constructor, which takes an instance of your `CameraHost` as a parameter.

Various sections below will mention using builder-style setters on `PictureTransaction` to control how a picture is taken.

`PictureTransaction` has a `tag(Object)` method that allow you to attach arbitrary data to the transaction, without having to bother with a subclass. You can retrieve that object later via the zero-argument `tag()` method.

Controlling the Names and Locations of Output Files

There are a series of methods that you can override on `SimpleCameraHost` to control where photos and videos are stored once taken. These methods will be called for each `takePicture()` or `startRecording()` call, so you can create customized results for each distinct photo or video.

Specifically:

- Override `getPhotoFilename()` to return the base name of the file to use to store the photo
- Override `getPhotoDirectory()` to return the name of the directory in which to store the photo
- Override `getPhotoPath()` to return the complete `File` object pointing to the desired file in the desired directory (the default implementation combines

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the results of `getPhotoDirectory()` and `getPhotoFilename()`, so overriding `getPhotoPath()` replaces all of that)

There are equivalent `getVideoFilename()`, `getVideoDirectory()`, and `getVideoPath()` for controlling the output of the next video to be taken.

`SimpleCameraHost.Builder` offers `photoDirectory()` and `videoDirectory()` setters, where you provide the `File` pointing to your desired directory.

By default, if you are using `SimpleCameraHost`, your image will be indexed by the `MediaStore`. If you do not want this, override `scanSavedImage()` to return `false` in your `SimpleCameraHost` subclass (or, call `scanSavedImage()` and pass in a `boolean` to use by default). This is called on a per-image basis.

Controlling Which Camera is Used

If you override `useFrontFacingCamera()` on `SimpleCameraHost` to return `true`, the front-facing camera will be used, instead of the default rear-facing camera. You can also call `useFrontFacingCamera()` on your `SimpleCameraHost.Builder`, passing in a `boolean` default value to use.

Or, override `getDeviceId()` (available on `CameraHost`), and you can provide the ID of the specific camera you want. This would involve your choosing an available camera based on your own criteria. See the JavaDocs for Android's `Camera` class, notably `getNumberOfCameras()` and `getCameraInfo()` for more. You can also call `deviceId()` on `SimpleCameraHost` to supply the camera ID to use.

Controlling FFC Mirror Correction

By default, the pictures taken from the front-facing camera are a mirror image of what is shown on the preview. If you wish for the front-facing camera photos to match the preview, override `mirrorFFC()` on your `CameraHost` and have it return `true`, and `CWAC-Camera` will reverse the image for you before saving it. Or, call `mirrorFFC()` on your `SimpleCameraHost.Builder`, supplying a `boolean` value to use as the default. Or, call `mirrorFFC()` on your `PictureTransaction`, to control this for an individual picture.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Handling Exceptions

There are some exceptions that are thrown by the `Camera` class (and kin, like `MediaRecorder`). Those are passed to your host's `handleException()` method. The default implementation displays a `Toast` and logs the message to LogCat as an error, but you probably will want to replace that with something else that integrates better with your UI.

Supporting "Full-Bleed Preview"

The original default behavior of `CameraFragment` and `CameraView` was to show the entire preview, as supplied by the underlying `Camera` API. Since the aspect ratio of the preview frames may be different than the aspect ratio of the `CameraView`, this results in a "letterbox" effect, where the background will show through on one axis on the sides.

The new default behavior is to completely fill the `CameraView`, at the cost of cropping off some of the actual preview frame, what is known as "full-bleed preview" (stealing some terminology from the world of print media).

To control this behavior:

- Have your `CameraHost` return `true` or `false` from `useFullBleedPreview()`
- Or, call `useFullBleedPreview()` on your `SimpleCameraHost.Builder`, passing in a `boolean` value to use by default.

Note that the pictures and videos taken by this library are unaffected by `useFullBleedPreview()`. Hence, if `useFullBleedPreview()` returns `true`, the picture or video may contain additional content on the edges that was not visible in the preview.

Wrapping the Preview UI

From a UI standpoint, the `CameraFragment` solely handles the preview pane. Presumably, you will need more to your UI than this, such as buttons to allow users to take pictures or record videos. You have two major options here:

1. You can put that UI as a peer to the `CameraFragment`, such as by having action bar items, as the demo apps do.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. You can subclass `CameraFragment` and override `onCreateView()`. Chain to the superclass to get the `CameraFragment`'s own UI, then wrap that in your own container with additional widgets, and return the combined UI from your `onCreateView()`. You can see this in the main demo app, which adds a `SeekBar` or `VerticalSeekBar` for zoom levels.

It is also possible to replace `onCreateView()` completely with your own implementation, or otherwise use `CameraView` from a layout resource. This is covered later in this document.

Auto-Focus

You can call `autoFocus()` on `CameraFragment` or `CameraView` to trigger any auto-focus behavior that you have configured via `setFocusMode()` on `Camera.Parameters`. You can call `cancelAutoFocus()` on `CameraFragment` or `CameraView` to ensure that auto-focus mode has been canceled.

Note that auto-focus is only available in certain conditions, notably when the preview mode is enabled. You can call `isAutoFocusAvailable()` on `CameraFragment` or `CameraView` to determine if auto-focus is presently available for use. Calling `autoFocus()` when auto-focus is not available will have no effect.

`CameraHost` implementations will need to implement an `onAutoFocus()` method, coming from the `Camera.AutoFocusCallback` interface that `CameraHost` extends. `SimpleCameraHost` has a default implementation of `onAutoFocus()` that plays a device-standard sound upon completion (API Level 16+ only).

`CameraHost` implementations will also need `autoFocusAvailable()` and `autoFocusUnavailable()` methods, to be notified when auto-focus is available or not. This can be used to trigger whether action bar items are enabled, etc. `SimpleCameraHost` has no-op implementations of these callbacks.

Single-Shot Mode

By default, the result of taking a picture is to return the `CameraFragment` to preview mode, ready to take the next picture. If, instead, you only need the one picture, or you want to send the user to some other bit of UI first and do not want preview to start up again right away, override `useSingleShotMode()` in your `CameraHost` to return `true`. Or, call `useSingleShotMode()` on your `SimpleCameraHost.Builder`, passing in a `boolean` to use by default. Or, call `useSingleShotMode()` on your `PictureTransaction`, to control this for an individual picture.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You will then probably want to use your own `saveImage()` implementation in your `CameraHost` to do whatever you want instead of restarting the preview. For example, you could start another activity to do something with the image. However, bear in mind that an `Intent` is limited to ~1MB, and so passing an image to another activity via an `Intent` extra is likely to be unreliable. You will need to do something else, such as (carefully) use a static data member.

Preview mode will re-enable automatically after an `onPause()`/`onResume()` cycle of your `CameraFragment`, or you can call `restartPreview()` on your `CameraFragment` (or `CameraView`).

Zoom Support

To zoom the camera, call `zoomTo()` on the `CameraView` or `CameraFragment`, supplying the integer zoom level that you want. This level must be between 0 and what `Camera.Parameters` returns from `getMaxZoom()`. The `adjustPreviewParameters()` callback method in your `CameraHost` is a good time to get this value and configure your UI (e.g., `SeekBar`) to allow the user to zoom the camera.

`zoomTo()` returns a `ZoomTransaction`. This has a series of builder-style methods (a.k.a., a fluent interface) that allow you to configure the transaction, where the methods return the transaction so you can chain on the next call. The configuration methods are:

- `onComplete()` to supply a `Runnable` to be executed when we have reached the zoom level
- `onChange()` to supply a `Camera.OnZoomChangeListener` to be called as we progress to the desired zoom level

Once configured, call `go()` to run the transaction.

If the camera supports smooth zoom, the zoom transaction will take a few moments, and you can cancel the operation by calling `cancel()` on the `ZoomTransaction`. If the camera does not support smooth zoom, the zoom level is just immediately changed.

Note that your `OnZoomChangeListener` supplied to `onChange()` will be called before the `onComplete()` `Runnable`, if you happen to supply both.

The main demo app adds a `SeekBar` and a `VerticalSeekBar` to control zoom levels, so you can see how this is used.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that some devices lie about their zoom capabilities. For example, the Motorola RAZR i's front-facing camera apparently does not support zoom, where `getMaxZoom()` still returns a positive value. `doesZoomReallyWork()` on your `CameraFragment` or `CameraView` will return `false` if zoom is known to be broken for the current camera on the current device. In this case, do not zoom, or your code may go "boom".

Camera? #FAIL

If `getCameraId()` of your `CameraHost` returns a negative value, `CameraView` will assume that there are no valid cameras (e.g., your app is running on a game console). In addition to avoiding anything that tries to touch the camera, your `CameraHost` will be called with `onCameraFail()`, where you will be supplied with a `FailureReason` of `NO_CAMERAS_REPORTED`.

If anything else goes wrong when trying to open the camera (e.g., a device admin policy has disabled the camera), your `onCameraFail()` method will be called with a `FailureReason` of `UNKNOWN`.

While `SimpleCameraHost` has a trivial `onCameraFail()` implementation (just logging to LogCat), you are strongly encouraged to override this and inform your users of the problem.

Fixing Up Images… And Your Heap

There are a few fixups that the library performs on your images, to provide consistent output. The biggest one is to rotate the image to the proper orientation, rather than rely on EXIF headers, as not all image viewers use those headers.

The problem is that these fixups take a lot of heap space. By default, the library will always try to perform these fixups, and for a high-resolution image on a low-memory device, an `OutOfMemoryError` may result.

You have two means of managing this, and you are welcome to apply one or both of them:

1. You can add `android:largeHeap="true"` to the `<application>` element of your manifest. On API Level 11+ devices, you will get more heap space, making it more likely that the fixup will succeed. However, this will hurt the user, in that your app will tend to kick other apps out of memory more quickly, which the user may not appreciate.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. Your `CameraHost` can return a value between `0.0f` and `1.0f` from `maxPictureCleanupHeapUsage()`. The default implementation on `SimpleCameraHost` returns `1.0f`, which says "the `byte` array of image data from the camera can be as big as our heap limit, and we should still try to do the cleanup work". A value of `0.0f` would indicate that the cleanup work should never be done, and the images will be saved in their natural state. A value in between represents a portion of the heap space; if the `byte` array is that size or smaller, go ahead and try to do the fixups. Note that this determination is made on the compressed JPEG `byte` array length, not the size of the decoded `Bitmap`, and the JPEG may be compressed ~90% compared to its uncompressed size.

Core Camera Concepts

This section will outline the basic concepts of working with the `Camera` and `MediaRecorder` APIs for simple operations: displaying a preview, taking a still picture, and recording some video. It will do so in the context of reviewing the CWAC-Camera demo application and some of the more advanced configuration options for working with `CameraFragment`.

The Permission and the Features

First, you need permission to use the camera. That way, when end users install your application off of the Internet, they will be notified that you intend to use the camera, so they can determine if they deem that appropriate for your application.

You simply need the `CAMERA` permission in your `AndroidManifest.xml` file, along with whatever other permissions your application logic might require.

Your manifest also should contain one or more `<uses-feature>` elements, declaring what you need in terms of camera hardware. By default, asking for the `CAMERA` permission indicates that you *need* a camera. More specifically, asking for the `CAMERA` permission indicates that you need an *auto-focus* camera.

If you plan to record video, using `MediaRecorder`, you will also want to request the `RECORD_AUDIO` permission, assuming that you were not intending to record a silent film.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A Camera is Optional

If you would like a camera, but having one is not essential for the use of your app, put the following `<uses-feature>` element in your manifest:

```
<uses-feature android:name="android.hardware.camera" android:required="false" />
```

This indicates that you would like a camera, but it is not required. This reverses the default established by the `CAMERA` permission.

A Camera is Required

Technically, you would not need any `<uses-feature>` element in your manifest to indicate that you need a camera, as the `CAMERA` permission would handle that for you. However, it is good form to explicitly declare it anyway:

```
<uses-feature android:name="android.hardware.camera" android:required="true" />
```

Not only does that make your manifest more self-documenting, but it also helps protect you in case the default behavior of the `CAMERA` permission changes.

Other Camera Features

There are three other camera features that you could consider having `<uses-feature>` elements for:

1. `android.hardware.camera.autofocus`, to indicate whether or not the device needs a camera with auto-focus capability.
2. `android.hardware.camera.flash`, to indicate whether or not the device must support a camera flash
3. `android.hardware.camera.front`, to indicate whether or not the app needs a front-facing camera specifically (`android.hardware.camera` requests a rear-facing camera)

Of these, the only one you should definitely include in your app is `android.hardware.camera.autofocus`, once again because of the default effects of requesting the `CAMERA` permission. In particular, if you do not absolutely need auto-focus capabilities, you can use `android:required="false"` to reverse the `CAMERA` default requirement.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What the Demo Uses

The CWAC-Camera demo app requests three permissions: `CAMERA`, `RECORD_AUDIO`, and `WRITE_EXTERNAL_STORAGE`. The latter permission is so that we can save our photos and videos to external storage.

The demo app also indicates that:

- A camera is required to use this app (`android.hardware.camera`)
- Neither a front-facing camera (`android.hardware.camera.front`) nor an auto-focus camera (`android.hardware.camera.autofocus`) is required

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.cwac.camera.demo"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="17"/>

  <uses-permission android:name="android.permission.CAMERA"/>
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
  <uses-permission android:name="android.permission.RECORD_AUDIO"/>

  <uses-feature
    android:name="android.hardware.camera"
    android:required="true"/>
  <uses-feature
    android:name="android.hardware.camera.front"
    android:required="false"/>
  <uses-feature
    android:name="android.hardware.camera.autofocus"
    android:required="false"/>

  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:largeHeap="true"
    android:theme="@style/AppTheme">
    <activity
      android:name="MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    </activity>
    <activity
      android:name="FullScreenActivity"
      android:theme="@android:style/Theme.Holo.Light.NoActionBar.Fullscreen">
    </activity>
    <activity
      android:name="DisplayActivity"
      android:label="@string/app_name">
    </activity>
  </application>

</manifest>
```

Warning: Do Not Use android.hardware.camera.any Yet

`android.hardware.camera.any` was added in API Level 17 (Android 4.2), to address the issues caused by the Nexus 7 and potential other future devices with a front-facing camera and no rear-facing camera. This feature would indicate that your app can work with *any* camera, not just a front-facing or rear-facing camera.

Alas, as of mid-December 2012, the Play Store did not support this specific `<uses-feature>` variant, causing your app to be unavailable to most devices.

Until there are signs that `android.hardware.camera.any` support has been added to the Play Store, you are best off avoiding this particular `<uses-feature>` element.

The Preview Surface

The camera preview is basically a stream of images, taken by the camera, usually at less than full resolution. Mostly, that stream is to be presented to the user on the screen, to help them "see what the camera sees", so they can line up the right picture.

For presenting the preview stream to the user, there are two typical solutions: `SurfaceView` and `TextureView`.

SurfaceView for the Camera

`SurfaceView` is used as a raw canvas for displaying all sorts of graphics outside of the realm of your ordinary widgets. In this case, Android knows how to display a live look at what the camera sees on a `SurfaceView`, to serve as a preview pane. A `SurfaceView` is also used for video playback, and a variation of `SurfaceView` called `GLSurfaceView` is used for OpenGL animations.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That being said, `SurfaceView` is a subclass of `View`, and so it can be added to your UI the same as any other widget:

- Include it in a layout
- Return it as the `View` from `onCreateView()` of a `Fragment`
- Instantiate it in Java and add it to some container via `addView()`
- Etc.

If your app will support API Level 10 and older, you will want to call `getSurfaceHolder().getType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS)` on the `SurfaceView`. A "push buffers" `SurfaceView` is one designed to have images pushed to the surface, usually from video playback or camera previews. A `SurfaceHolder` is a quasi-controller object for the `SurfaceView` — most interactions with the `SurfaceView` come by way of the `SurfaceHolder`. This bit of configuration is not needed on API Level 11 and higher, as Android handles it for us automatically as the `SurfaceView` is put to use.

TextureView for the Camera

`SurfaceView`, however, has some limitations. This is mostly tied back to the way it works, by "punching a hole" in the UI to allow some lower-level component (like the camera) to render stuff into it. While there is a transparent layer on top of this "hole", for use in alpha-compositing in any overlapping widgets, the `SurfaceView` *content* is not rendered as part of the normal view hierarchy. The net effect is that you cannot readily move, animate, or otherwise transform a `SurfaceView`.

`TextureView` was added in API Level 14 and works for camera previews as of API Level 15. `TextureView` serves much the same role as does `SurfaceView`, for showing camera previews, playing videos, or rendering OpenGL scenes. However, `TextureView` behaves as a regular `View` and so therefore can be animated and such without issue. However, to do this, `TextureView` relies upon hardware acceleration, so environments without hardware acceleration cannot use `TextureView`.

What CWAC-Camera Does

CWAC-Camera makes a decision of what to use for the preview pane based on the API level of the device (by default). On Android 4.1 and higher, CWAC-Camera will use a `TextureView`; otherwise, CWAC-Camera will use a `SurfaceView`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The actual logic for dealing with the preview surfaces is wrapped up in a `PreviewStrategy` class, with `SurfacePreviewStrategy` and `TexturePreviewStrategy` implementations. That way, if some device-specific tweaks are needed, the changes may be able to be isolated into some new `PreviewStrategy` flavor.

Obtaining and Initializing the Camera

Android devices may have zero, one, or more than one camera. Usually, "more than one" winds up being "two", a rear-facing camera and a front-facing camera. Usually, devices with only one camera have that camera be rear-facing, but not always (e.g., the Nexus 7 has only a front-facing camera).

Hence, you need to select what camera you want, then arrange to open and close access to that camera as needed.

Choosing a Camera

The simplest way to choose a camera is to not choose at all, and arrange to open the default camera. That default camera is the first rear-facing camera on the device. However, devices that have *no* rear-facing cameras effectively have no default camera, and so going with the default is rarely the correct choice.

Instead, you should iterate over the available cameras, to find the one that you want. Note, however, that this approach only works for API Level 9 and above — on older devices, you have no choice but to try to open the default camera.

To find out how many cameras there are for the current device, you can call the static `getNumberOfCameras()` method on the `Camera` class.

To find out details about a particular camera, you can call the static `getCameraInfo()` method on `Camera`. This takes two parameters:

- the ID of the camera to open, which will be a number from 0 to the number of available camera minus 1
- a `Camera.CameraInfo` object, into which `getCameraInfo()` will pour details about the camera

The most notable field on `Camera.CameraInfo` is `facing`, which tells you if this is a rear-facing (`Camera.CameraInfo.CAMERA_FACING_BACK`) or front-facing (`Camera.CameraInfo.CAMERA_FACING_FRONT`) camera.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In CWAC-Camera, the `CameraHost` interface requires a `getCameraId()` method implementation, which returns the camera ID that you choose. `SimpleCameraHost` chooses an implementation based upon the available cameras and what you return from `useFrontFacingCamera()`:

```
@Override
public int getCameraId() {
  if (cameraId == -1) {
    initCameraId();
  }

  return(cameraId);
}

private void initCameraId() {
  int count=Camera.getNumberOfCameras();
  int result=-1;

  if (count > 0) {
    result=0; // if we have a camera, default to this one

    Camera.CameraInfo info=new Camera.CameraInfo();

    for (int i=0; i < count; i++) {
      Camera.getCameraInfo(i, info);

      if (info.facing == Camera.CameraInfo.CAMERA_FACING_BACK
          && !useFrontFacingCamera()) {
        result=i;
        break;
      }
      else if (info.facing == Camera.CameraInfo.CAMERA_FACING_FRONT
          && useFrontFacingCamera()) {
        result=i;
        break;
      }
    }
  }

  cameraId=result;
}
```

If what you want is not available (e.g., you ask for a front-facing camera, and the device has none), we use the default camera ID (`0`).

Opening and Closing the Camera

Old code samples would open the camera by calling a zero-parameter static `open()` method on the `Camera` class. This opens the default camera, and as noted above, this

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

is rarely a good idea. However, it is your only option on API Level 8 and below, if you are still supporting such devices.

Instead, if you have the ID of the camera that you wish to open, call the one-parameter static `open()` method, passing in the ID of the camera.

Both flavors of `open()` return an instance of `Camera`, which you can hold onto in your activity or fragment that is working with the camera.

While you have access to this camera, no other process can. Hence, it is important to release the camera when you are no longer needing it. To release the camera, call `release()` on your `Camera` instance, after which it is no longer safe to use the camera. A common pattern is to `open()` the camera in `onStart()` or `onResume()` and `release()` it in `onPause()` or `onStop()`, so you tie up the camera only while you are in the foreground.

For example, `CameraView` — the class in CWAC-Camera that contains most of the smarts behind the two flavors of `CameraFragment` — opens the camera in `onResume()`, among other things:

```
  @TargetApi(Build.VERSION_CODES.ICE_CREAM_SANDWICH)
  public void onResume() {
    addView(previewStrategy.getWidget());

    if (camera == null) {
      cameraId=getHost().getCameraId();

      if (cameraId >= 0) {
        try {
          camera=Camera.open(cameraId);

          if (getActivity().getRequestedOrientation() !=
ActivityInfo.SCREEN_ORIENTATION_UNSPECIFIED) {
            onOrientationChange.enable();
          }

          setCameraDisplayOrientation();

          if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH
              && getHost() instanceof Camera.FaceDetectionListener) {

camera.setFaceDetectionListener((Camera.FaceDetectionListener)getHost());
          }
        }
        catch (Exception e) {
          getHost().onCameraFail(FailureReason.UNKNOWN);
        }
      }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      else {
        getHost().onCameraFail(FailureReason.NO_CAMERAS_REPORTED);
      }
    }
  }
```

Showing the Camera Preview

The user needs some sort of preview of what the camera lens sees, in order to line up a picture. In some cases, the preview is all that matters, such as so-called "mirror" apps that use the front-facing camera to show the user what they look like.

To successfully show a preview, you need to configure the preview, then have Android start (and stop) showing the preview on your supplied `SurfaceView` or `TextureView`.

Configuring the Preview

The biggest thing that we need to do to configure the preview is determine what size of preview images should be used. Devices cannot support arbitrary-sized previews. Instead, we need to ask the camera what preview sizes it supports, choose one, then configure the camera to use that specific preview size.

To do any of this, we need the `Camera.Parameters` associated with our chosen and open `Camera`. `Camera.Parameters` serves two roles:

- It tells us what is possible, in terms of camera capabilities, above and beyond the limited information reported by `Camera.Info`
- It is where we stipulate what behavior we want, by updating the parameters and associating the updated parameters with the `Camera`

Getting the `Camera.Parameters` object from a `Camera` is a simple matter of calling `getParameters()`.

To find out what the valid preview sizes are, we can call `getSupportedPreviewSizes()` on the `Camera.Parameters` object. This will return a `List` of `Camera.Size` objects, with each `Camera.Size` holding a `width` and a `height` as integers.

Choosing a preview size is a bit of an art form. The biggest constraint is that you do not want a preview size that is bigger than the space you have reserved for the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

preview area. Beyond that, it is up to you what to use. Whatever size you choose, you can pass to `setPreviewSize()` on the `Camera.Parameters`.

Then, you can call `setParameters()` on the `Camera`, passing in your modified `Camera.Parameters` object, to affect this change.

You will wind up with a block of code resembling:

```
Camera.Parameters parameters=camera.getParameters();
Camera.Size
previewSize=figureOutWhatPreviewSizeYouWant(parameters.getSupportedPreviewSizes());

parameters.setPreviewSize(previewSize.width, previewSize.height);

camera.setParameters(parameters);
```

(where you get to supply the `figureOutWhatPreviewSizeYouWant()` implementation)

In CWAC-Camera, your `CameraHost` will be called with `getPreviewSize()`, where you need to return a valid `Camera.Size` indicating the desired size of the preview frames. `getPreviewSize()` is passed:

- the display orientation, in degrees, with 0 indicating landscape, 90 indicating portrait, etc.
- the available width and height for the preview
- the `Camera.Parameters` object, from which you can determine the valid preview sizes by calling `getSupportedPreviewSizes()`

The `CameraUtils` class — also in the CWAC-Camera library – contains a pair of static methods with stock algorithms for choosing the preview size:

1. `getOptimalPreviewSize()` uses the algorithm found in the SDK camera sample app
2. `getBestAspectPreviewSize()` finds the preview size that most closely matches the aspect ratio of our available space

`SimpleCameraHost` uses `getBestAspectPreviewSize()` for the default implementation of `getPreviewSize()`. You can override `getPreviewSize()` and substitute in your own selection algorithm. Just make sure that the returned size is one of the ones returned by `getSupportedPreviewSizes()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, `SimpleCameraHost` also calls `mayUseForVideo()` on your subclass. If this returns `true` (the default), `SimpleCameraHost` calls `getPreferredPreviewSizeForVideo()` on `Camera.Parameters`, to get a preview size that will work for both still images and video. If you know that you will not be recording any video, you can override `mayUseForVideo()` to return `false`, and you may get a better preview size as a result.

Starting and Stopping the Preview

In principle, there are just three steps:

1. You attach your preview surface to the `Camera` by calling `setPreviewDisplay()` (if you are using a `SurfaceView`) or `setPreviewTexture()` (if you are using a `SurfaceTexture`)
2. You show the preview on-screen by calling `startPreview()` on the `Camera`
3. You stop showing the preview by calling `stopPreview()` on the `Camera`

However, timing is important.

You also cannot call `setPreviewDisplay()` or `startPreview()` before your preview surface is ready. To know when *that* is, you will need to register a listener with your surface:

- You can register a `SurfaceHolder.Callback` with the `SurfaceHolder` of your `SurfaceView` by calling `addCallback()` on the `SurfaceHolder`. Your `SurfaceHolder.Callback` will be called with `surfaceChanged()` when the surface is ready for use, at which point it is safe to call `setPreviewDisplay()` and `startPreview()`.
- You can register a `TextureView.SurfaceTextureListener` with your `TextureView` by means of the `setSurfaceTextureListener()` call. Your `TextureView.SurfaceTextureListener` will be called with `onSurfaceTextureAvailable()` at the point in time when it is safe to call `setPreviewTexture()` and `startPreview()`.

You also need to stop the preview before you `release()` the `Camera`. And, as we will see in the next section, you also need to restart your preview after taking a photo.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Taking a Photo

Taking a photo with a `Camera` is a matter of calling `takePicture()` on the `Camera` object. There are two flavors of `takePicture()`, for which three parameters are in common:

- a `Camera.ShutterCallback`, which will be called the moment the picture is taken, so that you can customize the "shutter" sound
- two `Camera.PictureCallback` objects, for raw (uncompressed) and JPEG photo data

The four-parameter version of `takePicture()` also takes a third `Camera.PictureCallback`, to be called when "a scaled, fully processed postview image is available". This explanation probably means something to somebody, but the author of this book has no idea what it means.

You cannot call `takePicture()` until *after* `startPreview()` has been called to set up a preview pane. `takePicture()` will automatically stop the preview. At some point, if you want to be able to take another photo, you will need to call `startPreview()` again. Note, though, that you *cannot* call `startPreview()` until after the final compressed photo has been delivered to your `Camera.PictureCallback` object.

Before you call `takePicture()`, you are going to want to adjust the `Camera.Parameters` to configure how the photo should be taken. The primary setting to adjust is the size of the picture to take. Just as you ask `Camera.Parameters` for available preview sizes and choose one, you can call `getSupportedPictureSizes()`, which returns a `List` of `Camera.Size` objects. You can then choose a size and pass its width and height to `setPictureSize()` on the `Camera.Parameters`. Other things to potentially adjust include:

- flash mode (`getSupportedFlashModes()` and `setFlashMode()`)
- focus mode (`getSupportedFocusModes()` and `setFocusMode()`)
- white balance (`getSupportedWhiteBalance()` and `setWhiteBalance()`)
- geo-tagging (`setGpsLatitude()`, `setGpsLongitude()`, `setGpsAltitude()`, etc.)
- JPEG image quality (`setJpegQuality()`)
- and so on

Your `CameraHost` will be called with `getPictureSize()`, for you to return the desired `Camera.Size` of the still images taken by the camera. You are simply passed the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`Camera.Parameters`, on which you can call `getSupportedPictureSizes()` to find out the possible picture sizes that you can choose from.

CWAC-Camera's `CameraUtils` class has a pair of methods for simple algorithms for choosing a picture size:

1. `getLargestPictureSize()` returns the `Camera.Size` that is the largest in area
2. `getSmallestPictureSize()` returns the `Camera.Size` that is the smallest in area

`SimpleCameraHost` uses `getLargestPictureSize()` for the default implementation of `getPictureSize()`. You can override `getPictureSize()` and substitute in your own selection algorithm. Just make sure that the returned size is one of the ones returned by `getSupportedPictureSizes()`.

Your `CameraHost` can also override `adjustPictureParameters()`, where you are passed a `Camera.Parameters` object and can alter it as needed for settings to apply to the photo that is about to be taken. In addition, you can override `getShutterCallback()` to return the `Camera.ShutterCallback` to use, where the default is `null` to use the device's default shutter sound.

The Rest of the Demo

Our subclass of `CameraFragment` is `DemoCameraFragment`. The reasons we are extending `CameraFragment` are:

- The fragment can support either a front-facing camera or a rear-facing camera, and we need to track which one this particular fragment instance is supposed to work with, by using the arguments `Bundle`
- Our fragment contributes items to the action bar to take a picture and to start/stop video recording
- We have a custom subclass of `SimpleCameraHost`, called `DemoCameraHost`, mostly for indicating whether or not we should be using the front-facing camera

```
package com.commonsware.cwac.camera.demo;

import android.annotation.TargetApi;
import android.content.Context;
import android.content.Intent;
import android.hardware.Camera;
import android.hardware.Camera.Face;
import android.hardware.Camera.Parameters;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.os.Bundle;
import android.os.SystemClock;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.widget.SeekBar;
import android.widget.SeekBar.OnSeekBarChangeListener;
import android.widget.Toast;
import com.commonsware.cwac.camera.CameraFragment;
import com.commonsware.cwac.camera.CameraHost;
import com.commonsware.cwac.camera.CameraUtils;
import com.commonsware.cwac.camera.SimpleCameraHost;
import com.commonsware.cwac.camera.PictureTransaction;

public class DemoCameraFragment extends CameraFragment implements
    OnSeekBarChangeListener {
  private static final String KEY_USE_FFC=
      "com.commonsware.cwac.camera.demo.USE_FFC";
  private MenuItem singleShotItem=null;
  private MenuItem autoFocusItem=null;
  private MenuItem takePictureItem=null;
  private MenuItem flashItem=null;
  private MenuItem recordItem=null;
  private MenuItem stopRecordItem=null;
  private MenuItem mirrorFFC=null;
  private boolean singleShotProcessing=false;
  private SeekBar zoom=null;
  private long lastFaceToast=0L;
  String flashMode=null;

  static DemoCameraFragment newInstance(boolean useFFC) {
    DemoCameraFragment f=new DemoCameraFragment();
    Bundle args=new Bundle();

    args.putBoolean(KEY_USE_FFC, useFFC);
    f.setArguments(args);

    return(f);
  }

  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    setHasOptionsMenu(true);

    SimpleCameraHost.Builder builder=
        new SimpleCameraHost.Builder(new DemoCameraHost(getActivity()));

    setHost(builder.useFullBleedPreview(true).build());
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
}

@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {
  View cameraView=
      super.onCreateView(inflater, container, savedInstanceState);
  View results=inflater.inflate(R.layout.fragment, container, false);

  ((ViewGroup)results.findViewById(R.id.camera)).addView(cameraView);
  zoom=(SeekBar)results.findViewById(R.id.zoom);
  zoom.setKeepScreenOn(true);

  setRecordingItemVisibility();

  return(results);
}

@Override
public void onPause() {
  super.onPause();

  getActivity().invalidateOptionsMenu();
}

@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  inflater.inflate(R.menu.camera, menu);

  takePictureItem=menu.findItem(R.id.camera);
  singleShotItem=menu.findItem(R.id.single_shot);
  singleShotItem.setChecked(getContract().isSingleShotMode());
  autoFocusItem=menu.findItem(R.id.autofocus);
  flashItem=menu.findItem(R.id.flash);
  recordItem=menu.findItem(R.id.record);
  stopRecordItem=menu.findItem(R.id.stop);
  mirrorFFC=menu.findItem(R.id.mirror_ffc);

  if (isRecording()) {
    recordItem.setVisible(false);
    stopRecordItem.setVisible(true);
    takePictureItem.setVisible(false);
  }

  setRecordingItemVisibility();
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.camera:
      takeSimplePicture();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      return(true);

    case R.id.record:
      try {
        record();
        getActivity().invalidateOptionsMenu();
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(),
              "Exception trying to record", e);
        Toast.makeText(getActivity(), e.getMessage(),
                       Toast.LENGTH_LONG).show();
      }

      return(true);

    case R.id.stop:
      try {
        stopRecording();
        getActivity().invalidateOptionsMenu();
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(),
              "Exception trying to stop recording", e);
        Toast.makeText(getActivity(), e.getMessage(),
                       Toast.LENGTH_LONG).show();
      }

      return(true);

    case R.id.autofocus:
      takePictureItem.setEnabled(false);
      autoFocus();

      return(true);

    case R.id.single_shot:
      item.setChecked(!item.isChecked());
      getContract().setSingleShotMode(item.isChecked());

      return(true);

    case R.id.show_zoom:
      item.setChecked(!item.isChecked());
      zoom.setVisibility(item.isChecked() ? View.VISIBLE : View.GONE);

      return(true);

    case R.id.flash:
    case R.id.mirror_ffc:
      item.setChecked(!item.isChecked());

      return(true);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  return(super.onOptionsItemSelected(item));
}

boolean isSingleShotProcessing() {
  return(singleShotProcessing);
}

@Override
public void onProgressChanged(SeekBar seekBar, int progress,
                              boolean fromUser) {
  if (fromUser) {
    zoom.setEnabled(false);
    zoomTo(zoom.getProgress()).onComplete(new Runnable() {
      @Override
      public void run() {
        zoom.setEnabled(true);
      }
    }).go();
  }
}

@Override
public void onStartTrackingTouch(SeekBar seekBar) {
  // ignore
}

@Override
public void onStopTrackingTouch(SeekBar seekBar) {
  // ignore
}

void setRecordingItemVisibility() {
  if (zoom != null && recordItem != null) {
    if (getDisplayOrientation() != 0
        && getDisplayOrientation() != 180) {
      recordItem.setVisible(false);
    }
  }
}

Contract getContract() {
  return((Contract)getActivity());
}

void takeSimplePicture() {
  if (singleShotItem!=null && singleShotItem.isChecked()) {
    singleShotProcessing=true;
    takePictureItem.setEnabled(false);
  }

  PictureTransaction xact=new PictureTransaction(getHost());

  if (flashItem!=null && flashItem.isChecked()) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      xact.flashMode(flashMode);
    }

    takePicture(xact);
  }

  interface Contract {
    boolean isSingleShotMode();

    void setSingleShotMode(boolean mode);
  }

  class DemoCameraHost extends SimpleCameraHost implements
      Camera.FaceDetectionListener {
    boolean supportsFaces=false;

    public DemoCameraHost(Context _ctxt) {
      super(_ctxt);
    }

    @Override
    public boolean useFrontFacingCamera() {
      if (getArguments() == null) {
        return(false);
      }

      return(getArguments().getBoolean(KEY_USE_FFC));
    }

    @Override
    public boolean useSingleShotMode() {
      if (singleShotItem == null) {
        return(false);
      }

      return(singleShotItem.isChecked());
    }

    @Override
    public void saveImage(PictureTransaction xact, byte[] image) {
      if (useSingleShotMode()) {
        singleShotProcessing=false;

        getActivity().runOnUiThread(new Runnable() {
          @Override
          public void run() {
            takePictureItem.setEnabled(true);
          }
        });

        DisplayActivity.imageToShow=image;
        startActivity(new Intent(getActivity(), DisplayActivity.class));
      }
      else {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        super.saveImage(xact, image);
      }
    }

    @Override
    public void autoFocusAvailable() {
      if (autoFocusItem != null) {
        autoFocusItem.setEnabled(true);

        if (supportsFaces)
          startFaceDetection();
      }
    }

    @Override
    public void autoFocusUnavailable() {
      if (autoFocusItem != null) {
        stopFaceDetection();

        if (supportsFaces)
          autoFocusItem.setEnabled(false);
      }
    }

    @Override
    public void onCameraFail(CameraHost.FailureReason reason) {
      super.onCameraFail(reason);

      Toast.makeText(getActivity(),
                     "Sorry, but you cannot use the camera now!",
                     Toast.LENGTH_LONG).show();
    }

    @Override
    public Parameters adjustPreviewParameters(Parameters parameters) {
      flashMode=
          CameraUtils.findBestFlashModeMatch(parameters,

Camera.Parameters.FLASH_MODE_RED_EYE,
                                             Camera.Parameters.FLASH_MODE_AUTO,
                                             Camera.Parameters.FLASH_MODE_ON);

      if (doesZoomReallyWork() && parameters.getMaxZoom() > 0) {
        zoom.setMax(parameters.getMaxZoom());
        zoom.setOnSeekBarChangeListener(DemoCameraFragment.this);
      }
      else {
        zoom.setEnabled(false);
      }

      if (parameters.getMaxNumDetectedFaces() > 0) {
        supportsFaces=true;
      }
      else {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        Toast.makeText(getActivity(),
                       "Face detection not available for this camera",
                       Toast.LENGTH_LONG).show();
      }

      return(super.adjustPreviewParameters(parameters));
    }

    @Override
    public void onFaceDetection(Face[] faces, Camera camera) {
      if (faces.length > 0) {
        long now=SystemClock.elapsedRealtime();

        if (now > lastFaceToast + 10000) {
          Toast.makeText(getActivity(), "I see your face!",
                         Toast.LENGTH_LONG).show();
          lastFaceToast=now;
        }
      }
    }

    @Override
    @TargetApi(16)
    public void onAutoFocus(boolean success, Camera camera) {
      super.onAutoFocus(success, camera);

      takePictureItem.setEnabled(true);
    }

    @Override
    public boolean mirrorFFC() {
      return(mirrorFFC.isChecked());
    }
  }
}
```

Most of this is fairly straight-forward, except perhaps the video recording action bar items, which we will examine later in this chapter.

Recording a Video

All of that was simply to set up the camera preview and to take a picture. To record a video, there is yet *more* work to be done, once again largely handled for you by `CameraFragment`.

Back in the chapter on recording audio, we saw `MediaRecorder` and how it can be used to record audio. `MediaRecorder` also records video, through a similar process. However, the particular recipe for recording video is very specific — doing things in the wrong order can easily screw up the recording.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The MediaRecorder Recipe

First, you need to set up the camera preview, just as you would for taking a still photo. After all, the user needs to have some idea of what is going to be recorded.

At the point in time of starting the recording, you need to `unlock()` the `Camera`. This basically says that you will want to start using the `Camera` again later, but you want to allow another process — specifically, the Stagefright process that implements `MediaRecorder` — to be able to use the `Camera` temporarily.

Then, after creating your `MediaRecorder` instance, you must do the following, in the stated sequence:

- Call `setCamera()` on `MediaRecorder`, handing it the just-unlocked `Camera`
- Call `setAudioSource()` on `MediaRecorder`, indicating where the audio for the recording should come from (typically `MediaRecorder.AudioSource.CAMCORDER`)
- Call `setVideoSource()` on `MediaRecorder`, indicating where the video to be recorded should come from (typically `MediaRecorder.VideoSource.CAMERA`)
- Call `setProfile()` on `MediaRecorder`, choosing one of the available recording profiles (e.g., `CamcorderProfile.get(cameraId, CamcorderProfile.QUALITY_HIGH)`)
- Call `setOutputFile()` on `MediaRecorder`, supplying the file path or `FileDescriptor` where the recorded video should be stored
- Call `setOrientationHint()` on `MediaRecorder`, indicating whether or not the camera is in portrait or landscape, so that information can be added to the metadata included in the video
- Call `setPreviewDisplay()` on `MediaRecorder`, to connect the `MediaRecorder` to the `SurfaceView` or `TextureView` that you are using for preview images, so it knows where to display the "preview" of what is being recorded
- Call `prepare()`, and then `start()`, on `MediaRecorder` to actually do the recording

Note that prior to Android 2.2, you cannot call `setProfile()`, but instead have to call a series of methods, like `setOutputFormat()` instead.

When you are done recording, you can call `stop()` and `release()` on the `MediaRecorder`, then call `reconnect()` on the `Camera` to return you to the state you were in prior to initiating the recording in the first place.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Recording in CWAC-Camera

`DemoCameraFragment` starts off with a visible/enabled `record` action bar item. When clicked, it calls `record()` on `CameraFragment` to begin the recording, then calls `invalidateOptionsMenu()` to refresh the action bar. Its `onCreateOptionsMenu()` takes into account whether or not it is recording (via the `isRecording()` method on `CameraFragment`) and makes the `stop` action bar item visible (and `record` invisible) when recording is going on. Tapping the `stop` action bar item triggers a call to `stopRecording()`, plus refreshes the action bar once again to switch the items back to normal mode.

```
  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.camera, menu);

    takePictureItem=menu.findItem(R.id.camera);
    singleShotItem=menu.findItem(R.id.single_shot);
    singleShotItem.setChecked(getContract().isSingleShotMode());
    autoFocusItem=menu.findItem(R.id.autofocus);
    flashItem=menu.findItem(R.id.flash);
    recordItem=menu.findItem(R.id.record);
    stopRecordItem=menu.findItem(R.id.stop);
    mirrorFFC=menu.findItem(R.id.mirror_ffc);

    if (isRecording()) {
      recordItem.setVisible(false);
      stopRecordItem.setVisible(true);
      takePictureItem.setVisible(false);
    }
```

`record()` on `CameraFragment` eventually routes to `record()` on `CameraView`, which goes through the aforementioned recipe:

```
  public void record() throws Exception {
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
      throw new UnsupportedOperationException(
                                            "Video recording supported only on
API Level 11+");
    }

    if (displayOrientation != 0 && displayOrientation != 180) {
      throw new UnsupportedOperationException(
                                            "Video recording supported only in
landscape");
    }

    Camera.Parameters pictureParams=camera.getParameters();

    setCameraPictureOrientation(pictureParams);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    camera.setParameters(pictureParams);

    stopPreview();
    camera.unlock();

    try {
      recorder=new MediaRecorder();
      recorder.setCamera(camera);
      getHost().configureRecorderAudio(cameraId, recorder);
      recorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
      getHost().configureRecorderProfile(cameraId, recorder);
      getHost().configureRecorderOutput(cameraId, recorder);
      recorder.setOrientationHint(outputOrientation);
      previewStrategy.attach(recorder);
      recorder.prepare();
      recorder.start();
    }
    catch (IOException e) {
      recorder.release();
      recorder=null;
      throw e;
    }
  }
```

`record()` uses three hooks supplied to `CameraHost` for configuring the recording:

- `configureRecorderAudio()`, where `SimpleCameraHost` uses `MediaRecorder.AudioSource.CAMCORDER`:

```
  @Override
  public void configureRecorderAudio(int cameraId,
                                     MediaRecorder recorder) {
    recorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
  }
```

- `configureRecorderProfile()`, where `SimpleCameraHost` uses the `QUALITY_HIGH` profile:

```
  @TargetApi(Build.VERSION_CODES.HONEYCOMB)
  @Override
  public void configureRecorderProfile(int cameraId,
                                       MediaRecorder recorder) {
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB
        || CamcorderProfile.hasProfile(cameraId,
                                       CamcorderProfile.QUALITY_HIGH)) {
      recorder.setProfile(CamcorderProfile.get(cameraId,
                                               CamcorderProfile.QUALITY_HIGH));
    }
    else if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB
        && CamcorderProfile.hasProfile(cameraId,
                                       CamcorderProfile.QUALITY_LOW)) {
      recorder.setProfile(CamcorderProfile.get(cameraId,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                                                CamcorderProfile.QUALITY_LOW));
    }
    else {
      throw new IllegalStateException(
                                        "cannot find valid CamcorderProfile");
    }
  }
```

- `configureRecorderOutput()`, where `SimpleCameraHost` calls `getVideoPath()` and uses that location for the output

```
  @Override
  public void configureRecorderOutput(int cameraId,
                                      MediaRecorder recorder) {
    recorder.setOutputFile(getVideoPath().getAbsolutePath());
  }
```

Advanced CWAC-Camera Features

`CameraFragment` and `CameraHost` provide other hooks as well, for you to use to further configure the way your app takes pictures and videos.

Controlling Preview Sizes

Your `CameraHost` will be called with `getRecordingHint()`, to determine if the preview should be optimized for possible video recording, or not (i.e., only still images will be taken). You can return a `CameraHost.RecordingHint` enum: `STILL_ONLY`, `VIDEO_ONLY`, or `ANY`.

Usually, your `CameraHost` will be called with `getPreviewSize()`, where you need to return a valid `Camera.Size` indicating the desired size of the preview frames. `getPreviewSize()` is passed:

- the display orientation, in degrees, with 0 indicating landscape, 90 indicating portrait, etc.
- the available width and height for the preview
- the `Camera.Parameters` object, from which you can determine the valid preview sizes by calling `getSupportedPreviewSizes()`

The `CameraUtils` class contains three static methods with stock algorithms for choosing the preview size:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. `getOptimalPreviewSize()` uses the algorithm found in the SDK camera sample app
2. `getBestAspectPreviewSize()` finds the preview size that most closely matches the aspect ratio of our available space
3. `getBestAspectPreviewSize(double)` finds the preview size that offers the biggest preview size that only differs from the desired aspect ratio by the supplied `closeEnough` value (`closeEnough` of `0.0d` would give the same results as does `getBestAspectPreviewSize()`)

`SimpleCameraHost` uses `getBestAspectPreviewSize()` for the default implementation of `getPreviewSize()`. You can override `getPreviewSize()` and substitute in your own selection algorithm. Just make sure that the returned size is one of the ones returned by `getSupportedPreviewSizes()`.

If `getRecordingHint()` returns `ANY` or `VIDEO_ONLY`, though, `CameraHost` supplies the preview size via `getPreferredPreviewSizeForVideo()` instead of `getPreviewSize()`. If you wish to use a different preview size for video, return it, otherwise return `null` and we will use the results from `getPreviewSize()` instead. `getPreferredPreviewSizeForVideo()` is passed a `Camera.Size` as a hint from the device for a value to use, instead of anything you might get yourself from `Camera.Parameters` — while using the hinted value is probably a good idea (if it is not `null`), it is not required.

Controlling Picture Sizes

Similarly, your `CameraHost` will be called with `getPictureSize()`, for you to return the desired `Camera.Size` of the still images taken by the camera. You are simply passed the `Camera.Parameters`, on which you can call `getSupportedPictureSizes()` to find out the possible picture sizes that you can choose from.

The `CameraUtils` class has a pair of methods for simple algorithms for choosing a picture size:

1. `getLargestPictureSize()` returns the `Camera.Size` that is the largest in area
2. `getSmallestPictureSize()` returns the `Camera.Size` that is the smallest in area

`SimpleCameraHost` uses `getLargestPictureSize()` for the default implementation of `getPictureSize()`. You can override `getPictureSize()` and substitute in your own selection algorithm. Just make sure that the returned size is one of the ones returned by `getSupportedPictureSizes()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Arbitrary Preview Configuration

When setting up the camera preview, your `CameraHost` will be called with `adjustPreviewParameters()`, passing in a `Camera.Parameters`. Here, you can make any desired adjustments to the camera preview, *except* the preview size (which you should be handling in `getPreviewSize()`). `adjustPreviewParameters()` returns the revised `Camera.Parameters`, where the stock implementation in `SimpleCameraHost` just returns the passed-in parameters unmodified.

Arbitrary Photo Configuration

Shortly after you call `takePicture()` on your `CameraFragment`, your `CameraHost` will be called with `adjustPictureParameters()`, passing in a `Camera.Parameters`. Here, you can make any desired adjustments to the parameters related to taking photos, *except* the image size (which you should be handling in `getPictureSize()`). `adjustPictureParameters()` returns the revised `Camera.Parameters`, where the stock implementation in `SimpleCameraHost` just returns the passed-in parameters unmodified.

Arbitrary Video Configuration

Shortly after you call `startRecording()`, your `CameraHost` will be called with:

- `configureRecorderAudio()`
- `configureRecorderProfile()`
- `configureRecorderOutput()`

in that order. Here, you can help tailor the way videos get recorded. Each of these is passed the ID of the camera being used for recording plus the `MediaRecorder` instance that does the actual recording.

The stock `SimpleCameraHost` does the following:

- In `configureRecorderAudio()`, `SimpleCameraHost` calls `setAudioSource(MediaRecorder.AudioSource.CAMCORDER)` on the `MediaRecorder`
- In `configureRecorderProfile()`, `SimpleCameraHost` calls `setProfile(CamcorderProfile.get(cameraId, CamcorderProfile.QUALITY_HIGH))` on the `MediaRecorder`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- In `configureRecorderOutput()`, `SimpleCameraHost` calls `setOutputFile(getVideoPath().getAbsolutePath())` on the `MediaRecorder` (where `getVideoPath()` was described earlier in this document)

While these are reasonable defaults, you are welcome to override these implementations to do something else.

Overriding Photo Saving

The default `SimpleCameraHost` logic for saving photos uses the `getPhotoPath()` and related methods discussed above. Actually saving the photo is done in `saveImage()`, called on your `CameraHost`, where `SimpleCameraHost` has a `saveImage()` implementation that writes the supplied `byte` array out to the desired location.

You are welcome to override `saveImage()` and do something else with the `byte` array, such as send it over the Internet. `saveImage()` is called on a background thread, so you do not have to do your own asynchronous work.

Another use for this is to find out when the saving is complete, so that you can use the resulting image. Just override `saveImage()`, chain to the superclass implementation, and when that returns, the image is ready for use.

There is also a `saveImage(PictureTransaction, Bitmap)` callback, giving you a decoded `Bitmap` instead of a `byte` array.

By default, the `saveImage()` that takes a `byte` array will be called, and not `saveImage(PictureTransaction, Bitmap)`. To change this, call `needBitmap(boolean)` and/or `needByteArray(boolean)` on your `PictureTransaction`, passing that `PictureTransaction` to `takePicture()`.

Note that if you say that you need the `Bitmap`, you are responsible for the `Bitmap` (e.g., calling `recycle()` on it) once it is handed to your host.

Controlling the Shutter Callback

You can subclass `PictureTransaction` and override `onShutter()` to do something when the shutter is pressed.

Also, your `CameraHost` implementation can return a `Camera.ShutterCallback` object via `getShutterCallback()`, which will be used in the underlying `takePicture()` call

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

on the Android `Camera`, giving you control to play a "shutter click" sound. `SimpleCameraHost` returns `null` from `getShutterCallback()`, to give you the device default behavior.

Detecting Faces

If you wish to use the face detection APIs available on API Level 14+, do the following:

1. Have your `CameraHost` implementation also implement `Camera.FaceDetectionListener`.
2. Override `adjustPreviewParameters()` in your `CameraHost` and take that opportunity to check the value of `getMaxNumDetectedFaces()`, a method on `Camera.Parameters`. If that returns 0, face detection is not supported by the device. **NOTE**: a better API for this may be added in the future.
3. Override `autoFocusAvailable()` in your `CameraHost`, and if face detection is enabled, call `startFaceDetection()` on your `CameraFragment` or `CameraView`. **NOTE**: a dedicated callback for this may be added in the future — this is a stop-gap to allow this fix to go in a patch release
4. Similarly, override `autoFocusUnavailable()` in your `CameraHost` and, if face detection is enabled, call `stopFaceDetection()` on your `CameraFragment` or `CameraView`. **NOTE**: a dedicated callback for this may be added in the future.

Note that this capability was added to version 0.5.1 of this library. Also note that, while you can safely call `startFaceDetection()` and `stopFaceDetection()` regardless of API level, `getMaxNumDetectedFaces()` should only be called on API Level 14+ devices, or you will be hit with a `VerifyError` or the equivalent.

Choosing a DeviceProfile

`CameraHost` exists to provide a hook for you to determine how your app should handle taking pictures and videos. `DeviceProfile`, on the other hand, provides information about how the *device* handles taking pictures and videos. Different devices do slightly different things when working with the camera. Sometimes this is based on API level, sometimes it is based on how the device manufacturer tinkered with Android, and sometimes it is based on the underlying camera hardware. `DeviceProfile` provides a place for the CWAC-Camera project to isolate these differences.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`CameraHost` has a `getDeviceProfile()` method that should return an instance of the `DeviceProfile` to use for the device that is running the app. The implementation of `getDeviceProfile()` on `SimpleCameraHost` calls the static `getInstance()` method on `DeviceProfile`, which chooses a `DeviceProfile` based on internal heuristics. If you encounter problems with certain devices, you can detect those in your `getDeviceProfile()` method and return a `DeviceProfile` that addresses your needs, otherwise settling for using the library's own choice of `DeviceProfile`.

Note that `SimpleCameraHost.Builder` also has a `deviceProfile()` setter method that you can call, passing in a `DeviceProfile` that will be used as the default, replacing the system default.

The stock `DeviceProfile` is largely driven by XML resources. These resources' names are of the form `cwac_camera_profile_XXX_YYY`, where `XXX` is the `Build.MANUFACTURER` and `YYY` is the `Build.MODEL`. Both `Build.MANUFACTURER` and `BUILD.MODEL` are converted to lowercase and have non-alphanumeric values converted to underscores, to ensure that we wind up with a valid resource filename. Each of those XML resource files has a `<deviceProfile>` root element, containing child elements for different values that can be overridden:

- `<doesZoomActuallyWork>` (a boolean, `true` or `false`) overrides the default zoom detection logic, with `false` meaning that the device lies and zoom is not really supported
- `<maxPictureHeight>` (an integer) is the largest number of pixels high to use for the camera picture; a `Camera.Size` taller than this is ignored
- `<minPictureHeight>` (an integer) is the smallest number of pixels high to use for the camera picture; a `Camera.Size` shorter than this is ignored
- `<pictureDelay>` (an integer) is a time in milliseconds to delay taking the picture after updating `Camera.Parameters` with the picture settings, for devices that seem to reset themselves when the parameters are updated, resulting in messed-up pictures
- `<portraitFFCFlipped>` (a boolean, `true` or `false`) indicates if the image cleanup work needs to flip the image if it was taken in portrait mode from the front-facing camera
- `<useDeviceOrientation>` (a boolean, `true` or `false`) indicates if we should skip the `setRotation()` call on `Camera.Parameters` due to a device bug, and instead should just use physical orientation in the image cleanup phase to get the picture to turn out right
- `<useTextureView>` (a boolean, `true` or `false`) overrides the default choice of whether to use a `SurfaceView` or a `TextureView` for the preview, normally driven by API level

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

So long as the resource exists with the right filename, the library should pick it up, so you can add ones in your app if needed.

Working Directly with CameraView

If you wish to eschew fragments, you are welcome to work with `CameraView` directly. To do this:

- Add it in Java code by calling its one-parameter constructor, taking your `Activity` as a parameter. At the present time, `CameraView` does not support being placed in a layout resource.
- Call `setHost()` on the `CameraView` as early as possible, to make sure that the `CameraView` is working with the right `CameraHost` implementation. Alternatively, override `getHost()` and return the right `CameraHost` there.
- Forward the `onResume()` and `onPause()` lifecycle events from your activity or fragment to the `CameraView`.

Otherwise, `CameraView` should work as a regular `View`... so long as you do not try to use it in a layout resource.

Using CameraView in a Layout Resource

If you want to use `CameraView` in a layout resource, you can, but your activity will need to implement the `CameraHostProvider` interface. This has one required method: `getCameraHost()`, which returns the `CameraHost` instance to be used with the `CameraView`. You would implement this in lieu of calling `setHost()` yourself.

If you want to take advantage of this and use your own layout in a `CameraFragment` subclass, simply override `onCreateView()` and do what you want. The only requirement, other than the `CameraHostProvider` mentioned above, is that your `onCreateView()` needs to call the `setCameraView()` method, supplying the `CameraView` instance to the superclass.

The `demo-layout/` directory contains a small sample project that demonstrates this technique.

Flash Modes

`CameraView`, as well as `CameraFragment`, has a `getFlashMode()` which returns the flash mode from `Camera.Parameters`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To adjust the flash mode, you can call `flashMode()` on your `PictureTransaction` to specify a mode to apply when the picture is taken. Or, call `setFlashMode()` on `CameraView` or `CameraFragment` when needed. Or, you can manually configure the `Camera.Parameters` object in `adjustPictureParameters()` and/or `adjustPreviewParameters()`.

The `CameraUtils` class has a `findBestFlashModeMatch()` method that takes a `Camera.Parameters` object, plus one or more `String` names of flash modes (e.g., `Camera.Parameters.FLASH_MODE_ON`), and returns the mode that appears first in your list of strings that is supported by the current camera.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Media Routes

Android can send audio and video to a variety of places, such as:

- Bluetooth headsets or headphones
- External displays, like a TV or monitor
- External devices that themselves play back media, such as a Chromecast

There is a common API for determining which of these "places" are available and allowing the user to choose which of these "places" should be used for a given bit of media. This common API centers around a `MediaRouter`, which is the focus of this chapter.

Prerequisites

Understanding this chapter requires that you have read the core chapters of the book. In addition, you should read the chapters on advanced action bar techniques and the AppCompat action bar backport.

Terminology

First, we need to establish some common ground in terms of..., well, terms.

Media

In this chapter, "media" refers to audio or video. This includes both media that may be stored on the device as well as media that may be streamed from some other source, frequently over the Internet.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Route

A route indicates where media should be played. There are three categories of routes that concern us:

- Where should we be playing live audio, in terms of speakers or headphones or other things connected to the device?
- Where should we be playing live video: on the device's own screen or on some other screen connected via a cable?
- Is there any sort of "remote playback" device available, such as a Chromecast, that can play back media on its own under our direction, rather than requiring our own app to play back the media itself

MediaRouter

`MediaRouter` is the name of a class (actually, two classes) that know what routes are possible given the current environment and what routes are selected for the different categories (by default or by user choice).

A Tale of Three MediaRouters

`MediaRouter` and its related classes represent a curious API. There are two versions of the `MediaRouter` class and up to *three* versions of some of the related support classes that will concern you as a developer.

android.media

`MediaRouter` debuted in Android in API Level 16, through classes added to the `android.media` package. This version of `MediaRouter` can work with live audio and live video routes, but not the Chromecast-style remote playback routes.

`android.media` also contains other classes that pertain to routes, such as `MediaRouteActionProvider`, a way to allow the user to choose media routes via an action bar item. The version of these classes in `android.media` work with native API Level 11 versions of the action bar and fragments.

android.support.v7.media

In 2013, an update to the Android Support package was released that contained another version of `MediaRouter` and kin, in `android.support.v7` packages. These are

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

contained in a dedicated Android library project that you can add to your app, found in the `extras/android/support/v7/mediarouter` directory of your Android SDK installation, if you have a current Android Support package installed.

While the native version of `MediaRouter` is a system service — obtained via `getSystemService()` – the `v7` version of `MediaRouter` is a singleton, obtained from a static `getInstance()` method on the `MediaRouter` class.

The good news is that this updated version of `MediaRouter` can work with all three categories of routes, including the Chromecast-style remote playback routes.

CWAC-MediaRouter

However, the bad news is that the `v7` version of `MediaRouter`'s support classes only support the Android Support backports of fragments and the action bar. This requires you to inherit from `ActionBarActivity` and use the `v4` version of `Fragment` and kin. This is a rather annoying limitation, considering that many developers have specifically started dropping support for older API levels to be able to *avoid* using this backport.

Fortunately, you have an alternative.

The author of this book has published the CWAC-MediaRouter library. This library contains a complete cross-port of the `mediarouter-v7` library, to remove the dependency upon `appcompat-v7` and support the native action bar.

Note that using CWAC-MediaRouter still requires the `mediarouter` Android library project, as the CWAC library only contains those classes that needed modification.

The samples in this chapter will illustrate the use of both Google's and CommonsWare's versions of the classes.

Attaching to MediaRouter

To be able to take advantage of all that `MediaRouter` has to offer, we need to obtain an instance of it and connect to that instance, via method calls and registering callbacks.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Getting a MediaRouter Instance

To get an instance of the `android.support.v7.media.MediaRouter` flavor of `MediaRouter`, call `getInstance()` on `MediaRouter`.

This is in contrast to the `android.media.MediaRouter` variant, which is a system service, obtained by calling `getSystemService()`.

Note that the `android.support.v7.media.MediaRouter` flavor is global for your process, but weakly held from a garbage collection standpoint. You need to ensure that you hold onto your instance of `MediaRouter` as long as you need it. Once your application code lets go of the `MediaRouter` instance, it becomes eligible for garbage collection, disposing of any registered callbacks and such along the way.

Working with Routes

`MediaRouter` has a `getSelectedRoute()` method that returns the media route chosen by the user, or the overall default if the user has not yet had a chance in your app to choose a route. This method returns a `MediaRouter.RouteInfo` object, containing details about the route. In particular, you can call `supportsControlCategory()` to determine if the route is a live audio route, a live video route, or a remote playback route, so you can take advantage of it accordingly.

There is also `getDefaultRoute()`, which, as the name suggests, returns the `MediaRouter.RouteInfo` instance that is the overall default for your app.

You can call `getRoutes()` to obtain a list of all routes known at the present time. You might use this to allow the user to choose a route, though `MediaRouteActionProvider` is generally a better choice, as will be seen later in this chapter.

Given that you have a `MediaRouter.RouteInfo` instance from somewhere, you can call `selectRoute()` to make this route the active one, replacing whatever the previously-selected route was.

Registering a Callback

You can also call `addCallback()` to provide a `MediaRouter.Callback` instance that will be invoked at various points in time based on the changes in media routes. `addCallback()` also takes a `MediaRouteSelector`, which describes what sorts of

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

routes you are interested in. We will examine `MediaRouteSelector` in greater detail in the coverage of `MediaRouteActionProvider` later in this chapter.

There are two flavors of `addCallback()`. Both take the `MediaRouteSelector` and the `MediaRouter.Callback`, but one also takes an `int` supplying flags to control the behavior of `addCallback()`. One flag of particular importance is `CALLBACK_FLAG_REQUEST_DISCOVERY`. This tells `MediaRouter` to not only set up the callback, but to attempt to find new routes previously unknown to it. Mostly, this is for remote callback routes, which require network I/O to find and are not necessarily known if not specifically scanned for.

`MediaRouter.Callback` is a class, not an interface. You create your own subclass of `MediaRouter.Callback` and override the callback methods that interest you. Some noteworthy callback methods include:

- `onRouteAdded()` and `onRouteRemoved()`, which are called when routes are newly detected or have been lost, such as when a user plugs in or unplugs an HDMI cable from the device
- `onRouteSelected()` is called when a new route is selected, either by the user (e.g., via `MediaRouteActionProvider`) or by you (e.g., via `selectRoute()`)
- `onRouteUnselected()` is also called when a new route is selected, but in this case, you are notified about the *old* route being unselected

When you are done with the callback, call `removeCallback()` on the `MediaRouter`, passing in the same `MediaRouter.Callback` instance you supplied to `addCallback()`.

We will see examples of using `MediaRouter.Callback` in the next section.

User Route Selection with MediaRouteActionProvider

To give the user some measure of control over where media is played, you can add a `MediaRouteActionProvider` to your action bar. This will add a button that, when tapped, will allow the user to choose routes of relevance to your app (live audio, live video, remote playback).

However, this does not really work the way you (or the user) might expect, simply because some routes are automatically applied by the OS. Depending upon what the Android device is connected to will determine what routes are automatically applied and which ones the user can choose via `MediaRouteActionProvider`. For example,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

while Android will route live video to an HDMI-connected external display automatically, the user must opt into connecting to a Chromecast for remote playback capability.

This section outlines how to use `MediaRouteActionProvider` — both the Google and CWAC versions — and what the user will see for various circumstances. Most of the sections will be focusing on the MediaRouter/ActionProvider sample project, which uses the Google version of `MediaRouteActionProvider`.

The Basic Project and Dependencies

The project has dependency on the `mediarouter` Android library project. Projects that need `mediarouter` will need to have access to the Android Support library from the SDK Manager and follow the instructions to add it to your project. Since `mediarouter` depends upon `appcompat`, you will need both library projects.

The AppCompat backport of the action bar requires that your activities use a theme extending from `Theme.AppCompat`. Hence, we have a `res/values/styles.xml` resource that defines `AppTheme` in the context of `Theme.AppCompat.Light.DarkActionBar`:

```
<resources>

  <style name="AppBaseTheme" parent="@style/
Theme.AppCompat.Light.DarkActionBar"></style>

  <style name="AppTheme" parent="AppBaseTheme">
    <!-- All customizations that are NOT specific to a particular API-level can
go here. -->
  </style>

</resources>
```

And our `<activity>` in the manifest, pointing to `MainActivity`, refers to that theme:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.mrap"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="18"/>

  <application
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name="com.commonsware.android.mrap.MainActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

The Menu Resource

Since `MediaRouteActionProvider` is an action provider, we can to add it to our action bar via an `actionProviderClass` attribute in a menu resource. And, since the Google implementation of `MediaRouteActionProvider` works with the AppCompat action bar backport, we specifically need to use the AppCompat approach to adding `actionProviderClass`, putting it in our app's custom XML namespace:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <item
    android:id="@+id/route_provider"
    android:title="@string/route_provider_title"
    app:actionProviderClass="android.support.v7.app.MediaRouteActionProvider"
    app:showAsAction="always"/>

</menu>
```

Initializing the MediaRouter and Selector

Our activity (`MainActivity`) is an `ActionBarActivity` subclass, following the rules for using the AppCompat action bar backport:

```
package com.commonsware.android.mrap;

import android.os.Bundle;
import android.support.v4.view.MenuItemCompat;
import android.support.v7.app.ActionBarActivity;
import android.support.v7.app.MediaRouteActionProvider;
import android.support.v7.media.MediaControlIntent;
import android.support.v7.media.MediaRouteSelector;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.support.v7.media.MediaRouter;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.TextView;

public class MainActivity extends ActionBarActivity {
  private MediaRouteSelector selector=null;
  private MediaRouter router=null;
  private TextView selectedRoute=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    selectedRoute=(TextView)findViewById(R.id.selected_route);

    router=MediaRouter.getInstance(this);
    selector=
        new
MediaRouteSelector.Builder().addControlCategory(MediaControlIntent.CATEGORY_LIVE_AUDIO)

.addControlCategory(MediaControlIntent.CATEGORY_LIVE_VIDEO)

.addControlCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK)
                                        .build();

  }

  @Override
  public void onResume() {
    super.onResume();

    router.addCallback(selector, cb,
                       MediaRouter.CALLBACK_FLAG_REQUEST_DISCOVERY);
  }

  @Override
  public void onPause() {
    router.removeCallback(cb);

    super.onPause();
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);

    MenuItem item=menu.findItem(R.id.route_provider);
    MediaRouteActionProvider provider=
        (MediaRouteActionProvider)MenuItemCompat.getActionProvider(item);

    provider.setRouteSelector(selector);

    return(true);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }

  private MediaRouter.Callback cb=new MediaRouter.Callback() {
    @Override
    public void onRouteSelected(MediaRouter router,
                                MediaRouter.RouteInfo route) {
      selectedRoute.setText(route.toString());
    }
  };
}
```

In `onCreate()` we obtain an instance of `MediaRouter`. More specifically, we obtain an instance of `android.support.v7.media.MediaRouter`.

We also will need a `MediaRouteSelector` instance. `MediaRouteSelector` expresses rules for what sorts of media routes we are interested in. The simplest way to set up a `MediaRouteSelector` is to use the `MediaRouteSelector.Builder` inner class, which follows the fluent API style of other Android `Builder` classes (e.g., `Notification.Builder`, `AlertDialog.Builder`). Here, we call `addControlCategory()` three times, indicating three categories of routes that we are interested in:

- `MediaControlIntent.CATEGORY_LIVE_AUDIO`
- `MediaControlIntent.CATEGORY_LIVE_VIDEO`
- `MediaControlIntent.CATEGORY_REMOTE_PLAYBACK`

Calling `build()` on the resulting `Builder` gives us our `MediaRouteSelector`, which we will use elsewhere in the activity.

Configuring the ActionProvider

In `onCreateOptionsMenu()` of `MainActivity`, we inflate our menu resource and pull out the `MediaRouteActionProvider`. To obtain an action provider from the AppCompat action bar, the simplest solution is to use the `MenuItemCompat` helper class from the Android Support package, calling its static `getActionProvider()` method. This will work both with the AppCompat backport of the action bar *and* with the native API Level 11+ action bar, though you do not need to use `MenuItemCompat` for the latter if you do not want.

We then call the `setRouteSelector()` method on our `MediaRouteActionProvider` instance, passing in the `MediaRouteSelector` we configured back in `onCreate()`. This tells the action provider what routes the user should be able to configure. In

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

our case, that is all three major categories of routes (live audio, live video, and remote playback).

Registering for Route Changes

Interestingly enough, that is insufficient to make the `MediaRouteActionProvider` work. We *also* need to register a `MediaRouter.Callback` with the `MediaRouter`, to be informed about events related to media routes. Our `cb` private data member is an instance of an anonymous inner class extending `MediaRouter.Callback`, overriding the `onRouteSelected()` method. This method will be called whenever a new route is selected, telling us the `MediaRouter.RouteInfo` of the newly-selected route. In our case, we just update a `TextView` that is our activity's UI with the details of that route, courtesy of calling `toString()` on the `RouteInfo` object.

To inform `MediaRouter` about our desire for such callbacks, we need to call `addCallback()` on the `MediaRouter`, and later on call `removeCallback()` when we no longer need to know about such events. In `MainActivity`, these steps are done in `onResume()` and `onPause()`, respectively.

Note that we provide the `CALLBACK_FLAG_REQUEST_DISCOVERY` flag in the `addCallback()` method, to trigger a search for any Chromecast or other remote playback-capable devices that can serve as media routes.

The Results

Running this on an emulator is largely pointless, as emulators do not emulate media routes.

Running this on a device will give varying results, depending upon what other media-related accessories are available to that device. If there are no user-selectable media routes available, the `MediaRouteActionProvider` is marked as invisible, so the user does not see the icon and perhaps get confused by why tapping on it has no effect.

However, our `TextView` will show some initial route that was chosen by the device:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

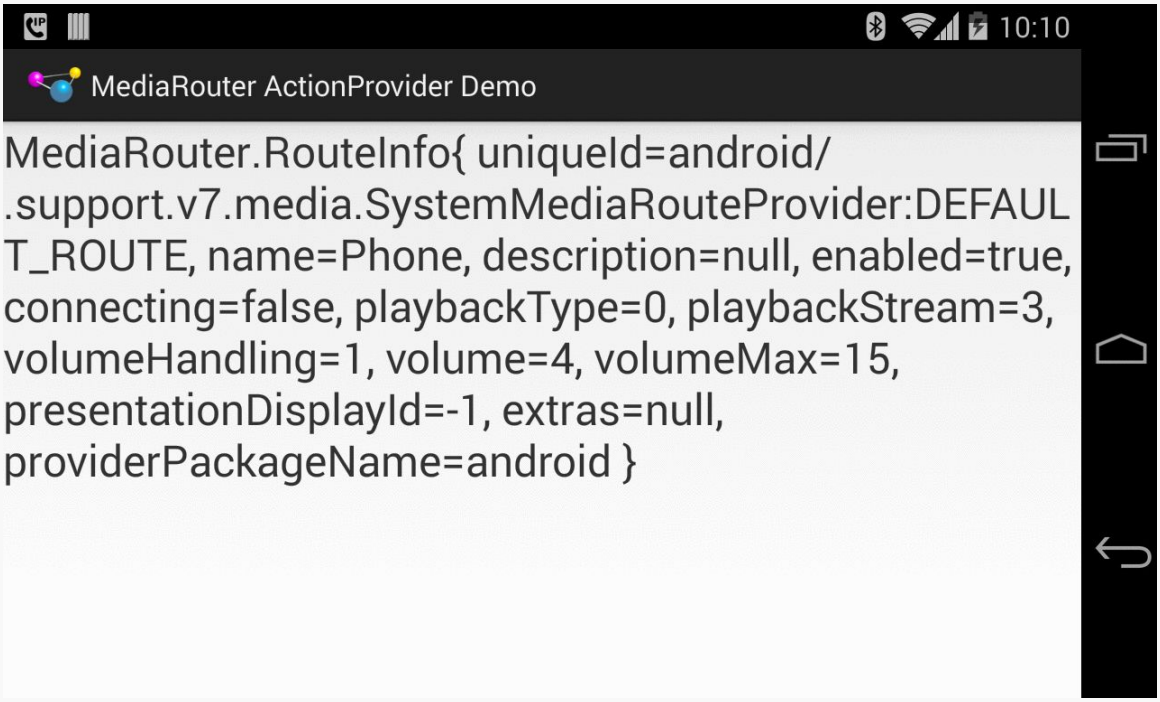


Figure 619: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Default Route

Live Audio Routes

If you launch the demo with some form of external headset or speakers attached, such as via Bluetooth, you will see the route for that is automatically selected:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

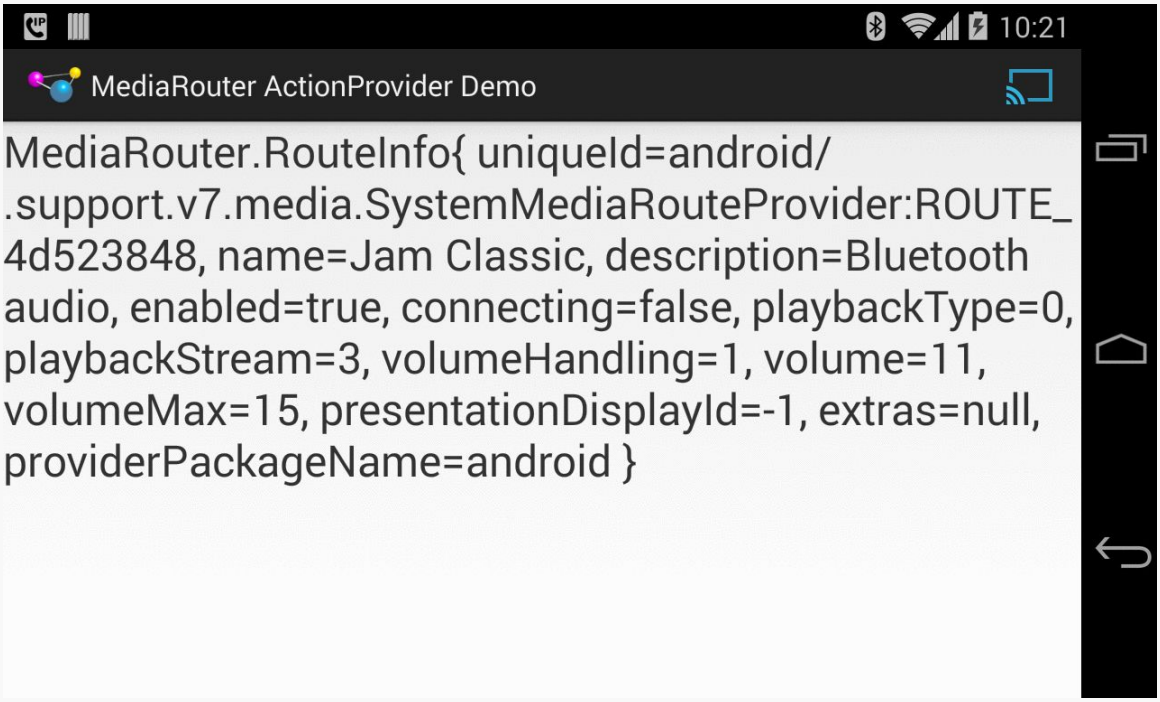


Figure 620: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Live Audio Route

The `MediaRouteActionProvider` appears, with a blue highlight, indicating an active selected route. More importantly, the blue highlight indicates that the route is configurable by tapping on it to bring up a dialog:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

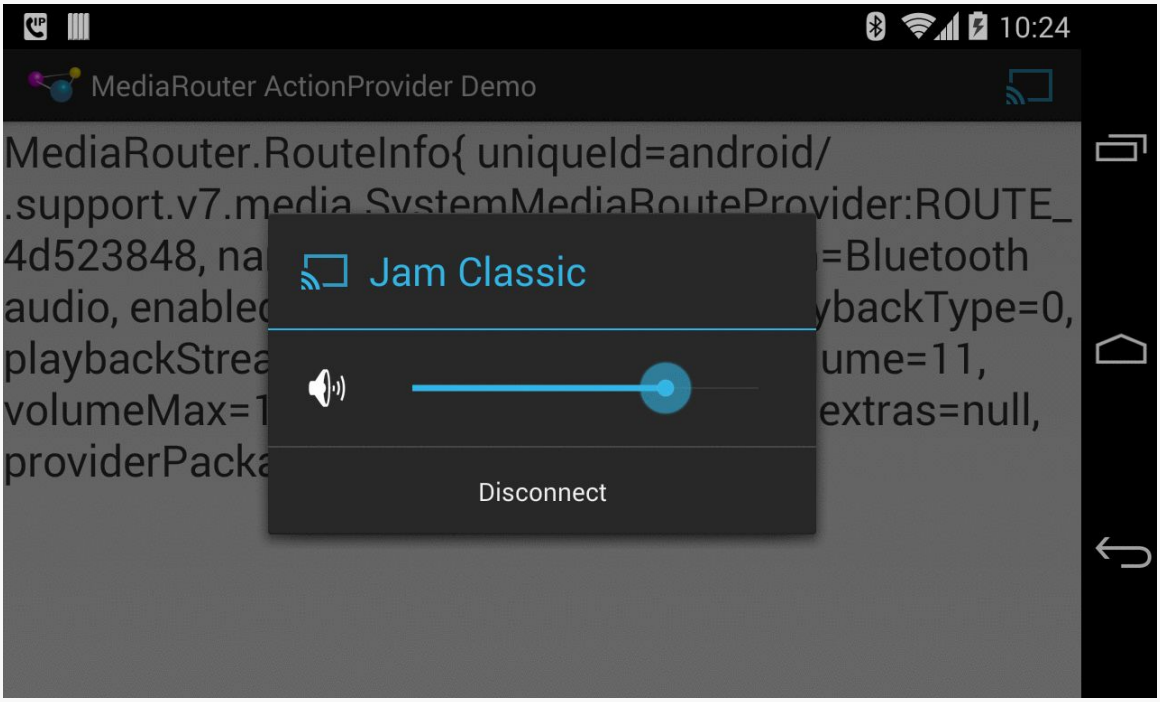


Figure 621: MediaRouter ActionProvider Demo, on a Nexus 4, Live Audio Route Configuration

Here, we can adjust the volume, plus disconnect from the route. Disconnecting shows our `MediaRouteActionProvider` with the default white highlight:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

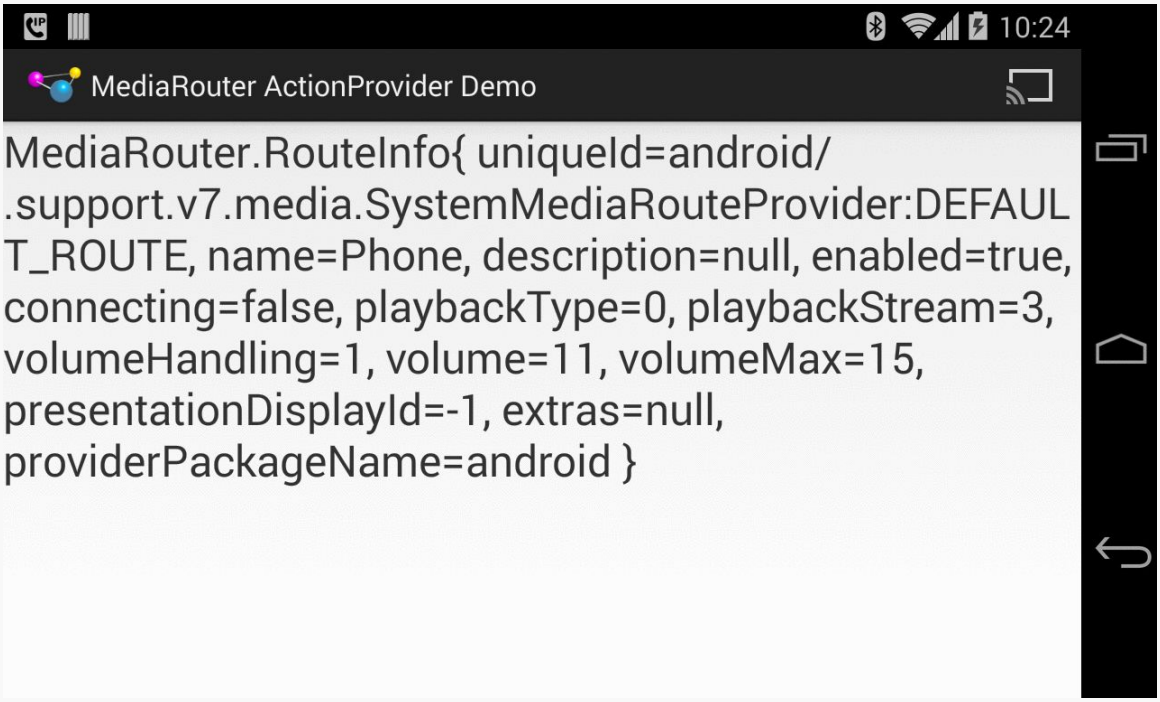


Figure 622: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Default Route and Provider

The white highlight means that there are possible routes, though none in use. Tapping the icon brings up a connection dialog:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

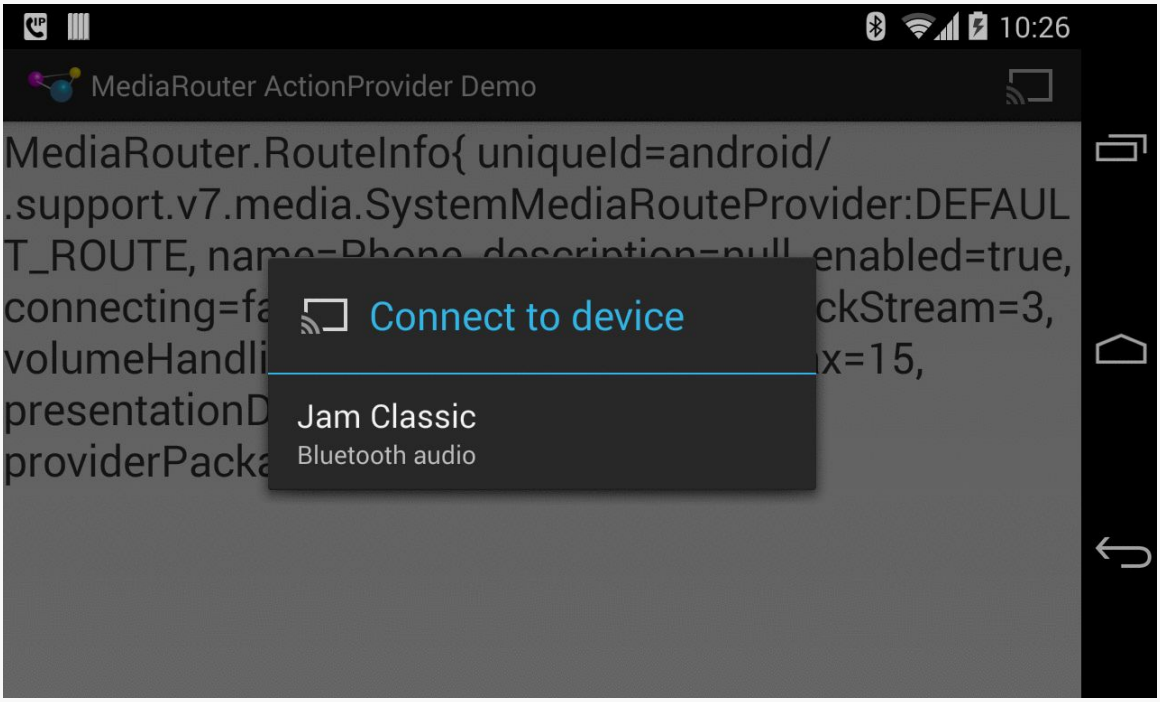


Figure 623: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Available Routes

Live Video Routes

If you launch the demo with some form of external display attached — HDMI, MHL, SlimPort, etc. — you still will not see the `MediaRouteActionProvider`, as live video routes are automatically selected, at least if there is only one such route.

However, `onRouteSelected()` will still be called as part of starting up the activity, so the `TextView` will reflect the live video route:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

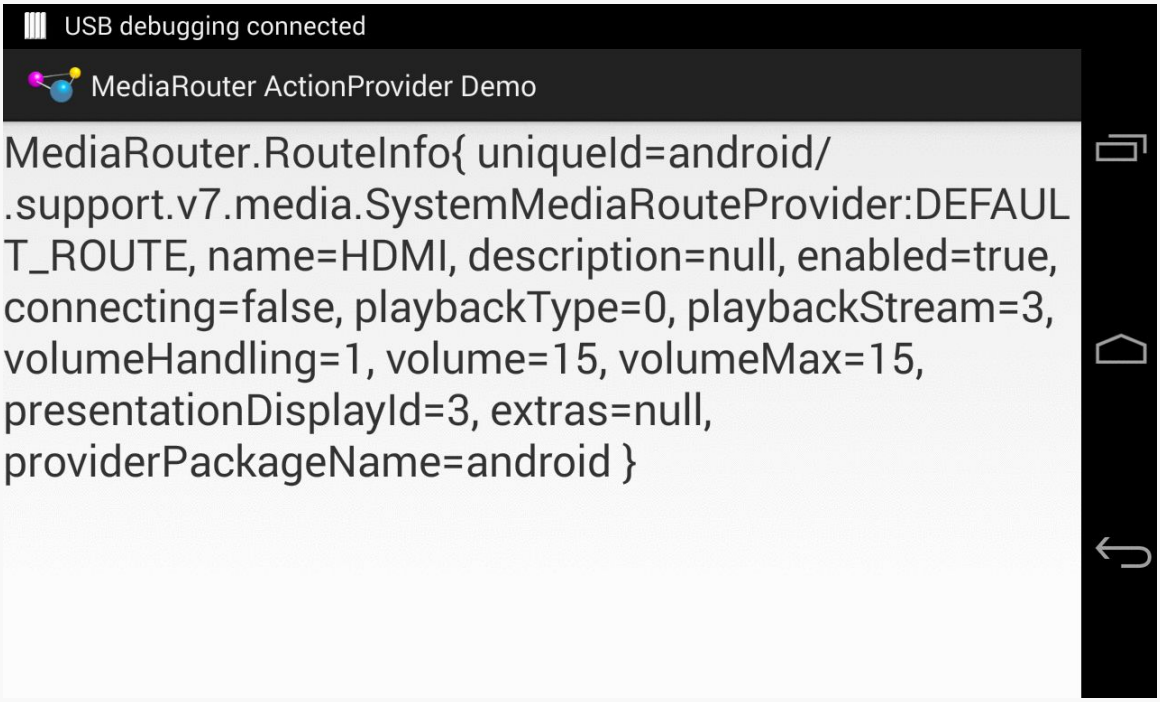


Figure 624: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Live Video Route

Remote Playback Routes

Since the user has to opt into remote playback media routes, the `MediaRouteActionProvider` *will* appear if you configure it to show such routes and a route is available:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

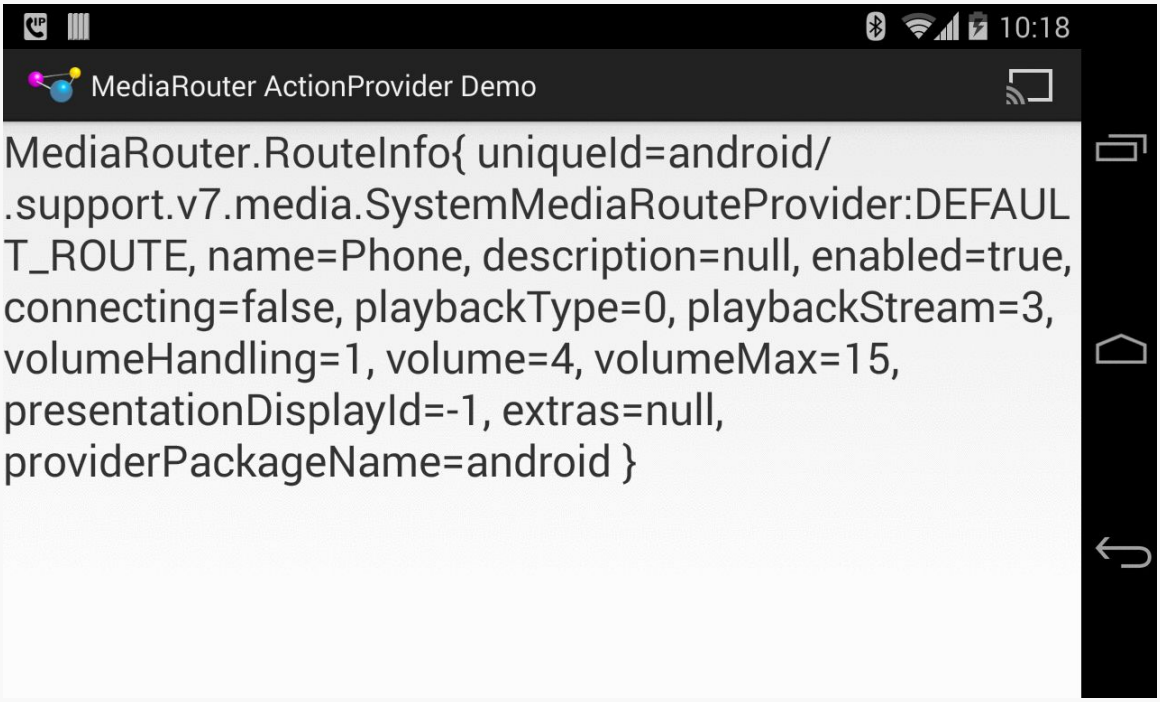


Figure 625: MediaRouter ActionProvider Demo, on a Nexus 4, Showing ActionProvider

The `MediaRouteActionProvider`, when tapped, will pop up a dialog of available routes that the user can select:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

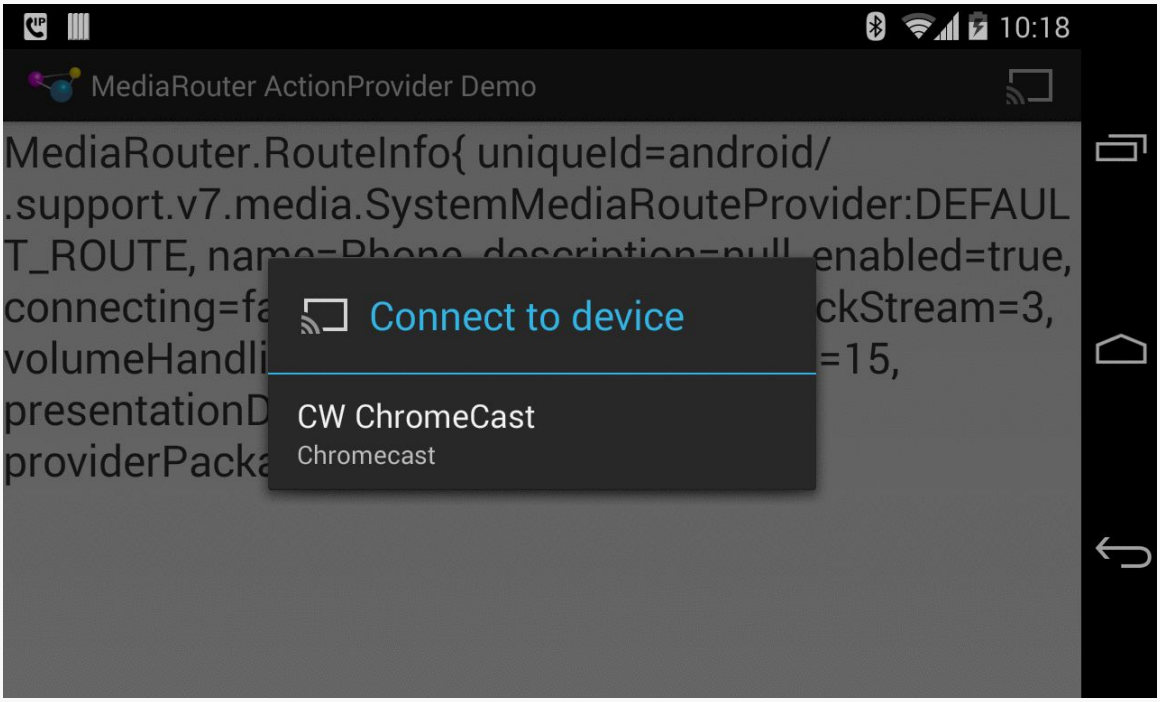


Figure 626: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Available Chromecast Route

Note that if the device has both a Bluetooth audio connection *and* access to a remote playback route (like a Chromecast), and you requested both live audio and remote playback routes, then the route selection dialog could have multiple choices:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

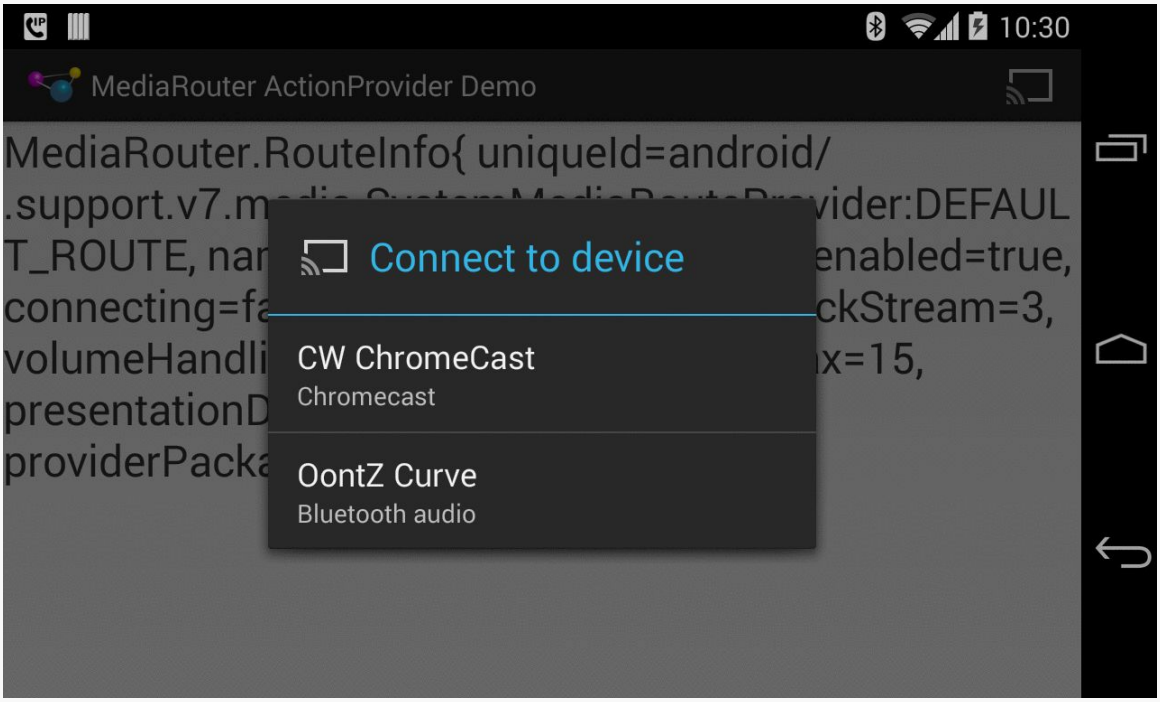


Figure 627: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Multiple Available Routes

If the user chooses a route from the dialog, our `onRouteSelected()` method will be called to reflect the new selection:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

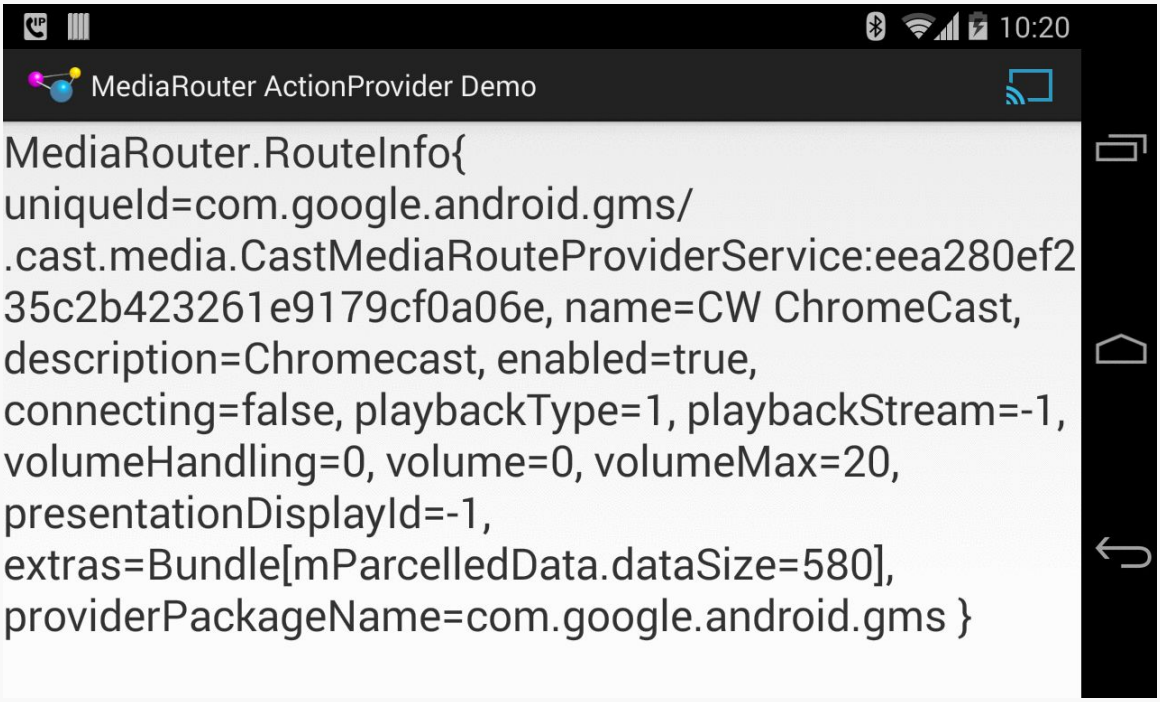


Figure 628: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Selected Chromecast Route

Also note that the `MediaRouteActionProvider` color changes from white to blue, indicating an altered route.

Tapping the action provider again pops up a dialog to control the volume of the route, plus a "Disconnect" button:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

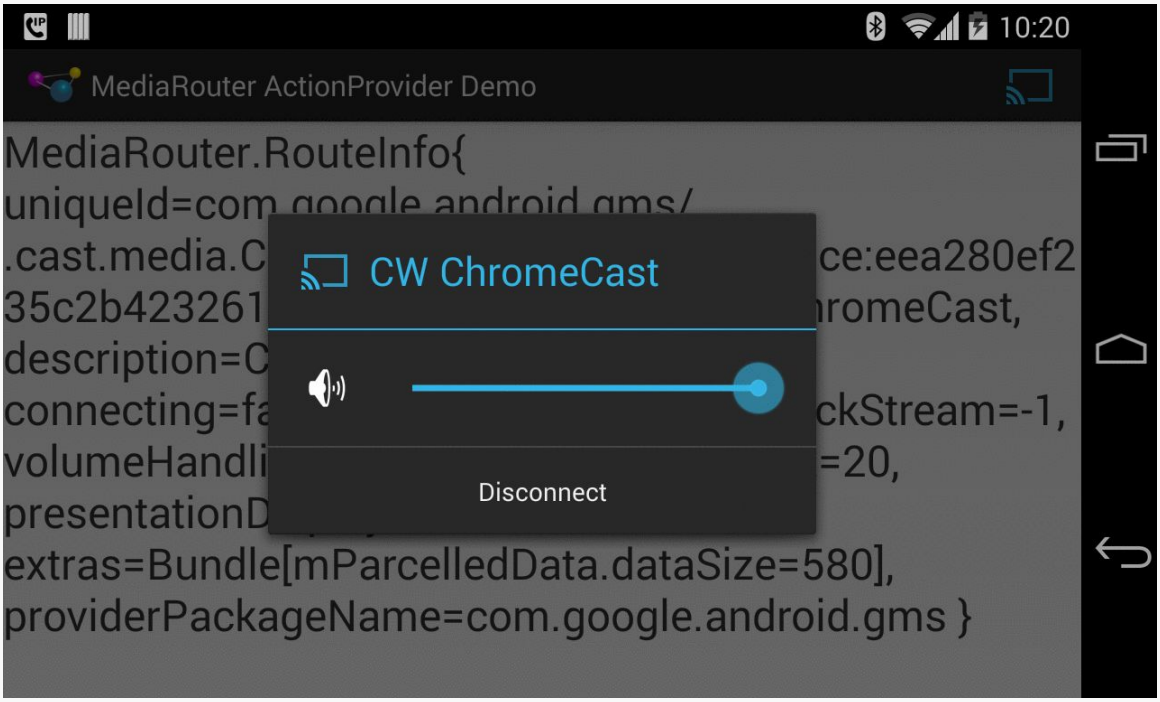


Figure 629: MediaRouter ActionProvider Demo, on a Nexus 4, Showing Route Dialog

Tapping that "Disconnect" button returns everything to its original state.

The CWAC-MediaRouter Version

If you wish to use the regular `Activity` class, in conjunction with native API Level 11+ versions of the action bar and fragments, you can have nearly identical code by means of the CWAC-MediaRouter library. With that library added to your project, you can remove the AppCompat code from your app, such as switching to a regular theme (e.g., `Theme.Holo.Light.DarkActionBar`) and inheriting from `Activity` instead of `ActionBarActivity`.

However, you will need to use `com.commonsware.cwac.mediarouter.MediaRouteActionProvider` as your `android:actionProvider` in your menu resource:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/route_provider"
    android:actionProviderClass=
      "com.commonsware.cwac.mediarouter.app.MediaRouteActionProvider"
    android:showAsAction="always"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:title="@string/route_provider_title"/>

</menu>
```

Similarly, your Java code will need to import `com.commonsware.cwac.mediarouter.MediaRouteActionProvider` instead of `android.support.v7.app.MediaRouteActionProvider`. However, you will still use the `mediarouter-v7` versions of the other `MediaRouter`-related classes, such as `MediaRouter` itself. The CWAC-MediaRouter library only contains ports for those classes that depended upon the backported action bar and fragments.

Otherwise, though, your code would be unchanged from the version that uses `mediarouter-v7` for everything, yet your `MediaRouteActionProvider` will work using native API Level 11+ fragments and the action bar.

Using Live Audio Routes

Using live audio routes will be covered in a future edition of this book.

Using Live Video Routes

A live video route is designed to be used with `Presentation`, a class that enables you to render your own content on the external display, much like how you would render your own content in a `Dialog`.

The use of `Presentation` is covered in an upcoming chapter.

Using Remote Playback Routes

In principle, `RemotePlaybackClient` allows you to work with remote playback routes, to specify `Uri` values to play back.

In practice, not even Google's own sample code for `RemotePlaybackClient` works reliably, let alone as documented.

That being said, let's take a look at the MediaRouter/RemotePlayback sample project, to see how `RemotePlaybackClient` works and where the current problems lie.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Setting Up MediaRouteActionProvider

Much of the basic setup of this application mirrors the `MediaRouteActionProvider` sample shown earlier in this chapter. One difference is that the UI is now encapsulated in a `PlaybackFragment`, with `MainActivity` simply setting up that fragment when needed:

```
package com.commonsware.android.remoteplayback;

import android.os.Bundle;
import android.support.v7.app.ActionBarActivity;

public class MainActivity extends ActionBarActivity {
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (getSupportFragmentManager().findFragmentById(android.R.id.content) ==
null) {
      getSupportFragmentManager().beginTransaction()
                                 .add(android.R.id.content,
                                      new PlaybackFragment()).commit();
    }
  }
}
```

Note that this sample does not use the CWAC-MediaRouter project, so `MainActivity` is an `ActionBarActivity`, and so on.

`PlaybackFragment`, when it is created, opts into being retained on configuration changes, tells Android that it wishes to add items to the action bar, and sets up a `MediaRouteSelector` for `CATEGORY_REMOTE_PLAYBACK` routes:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setRetainInstance(true);
    setHasOptionsMenu(true);
    selector=
        new MediaRouteSelector.Builder()

.addControlCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK).build();
  }
```

Then, in `onAttach()` — called when the `PlaybackFragment` is attached to the hosting activity — we obtain a `MediaRouter` instance:

```
  @Override
  public void onAttach(Activity host) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onAttach(host);

    router=MediaRouter.getInstance(host);
  }
```

In `onResume()`, we hook a `cb` data member — an instance of `MediaRouter.Callback` up to the `MediaRouter`, also requesting that the `MediaRouter` initiate discovery of available routes. We remove our callback in `onPause()`:

```
  @Override
  public void onResume() {
    super.onResume();

    router.addCallback(selector, cb,
                       MediaRouter.CALLBACK_FLAG_REQUEST_DISCOVERY);
  }

  @Override
  public void onPause() {
    router.removeCallback(cb);

    super.onPause();
  }
```

We will examine `cb`'s declaration later in this section.

Later on, as part of our `onCreateOptionsMenu()` processing, we configure the `MediaRouteActionProvider` as before:

```
    MenuItem item=menu.findItem(R.id.route_provider);
    MediaRouteActionProvider provider=
        (MediaRouteActionProvider)MenuItemCompat.getActionProvider(item);

    provider.setRouteSelector(selector);
```

All of this is very similar to the earlier examples. From here, though, we will actually use the route once the user selects it, to play back some media.

The Rest of the User Interface

The UI of the `PlaybackFragment` — other than the action bar — consists of a "transcript". This is a `TextView` inside of a `ScrollView`:

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TextView
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:id="@+id/transcript"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="20sp"/>

</ScrollView>
```

As with most fragments, we inflate this layout in `onCreateView()`, holding onto the `TextView` and `ScrollView` widgets:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    scroll=
        (ScrollView)inflater.inflate(R.layout.activity_main, container,
                                     false);

    transcript=(TextView)scroll.findViewById(R.id.transcript);

    logToTranscript("Started");

    return(scroll);
  }
```

The `logToTranscript()` method will append a `String` to the `TextView` contents on a new line, plus scroll to the bottom to ensure that the new text is visible:

```
  private void logToTranscript(String msg) {
    if (client != null) {
      String sessionId=client.getSessionId();

      if (sessionId != null) {
        msg="(" + sessionId + ") " + msg;
      }
    }

    transcript.setText(transcript.getText().toString() + msg + "\n");
    scroll.fullScroll(View.FOCUS_DOWN);
  }
```

The `client` data member referred to in `logToTranscript()` is our `RemotePlaybackClient` instance, which will be covered in the next section.

What the user sees when first running the sample is the action bar (with our `MediaRouteActionProvider`) and the transcript, with a simple “Started” message:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

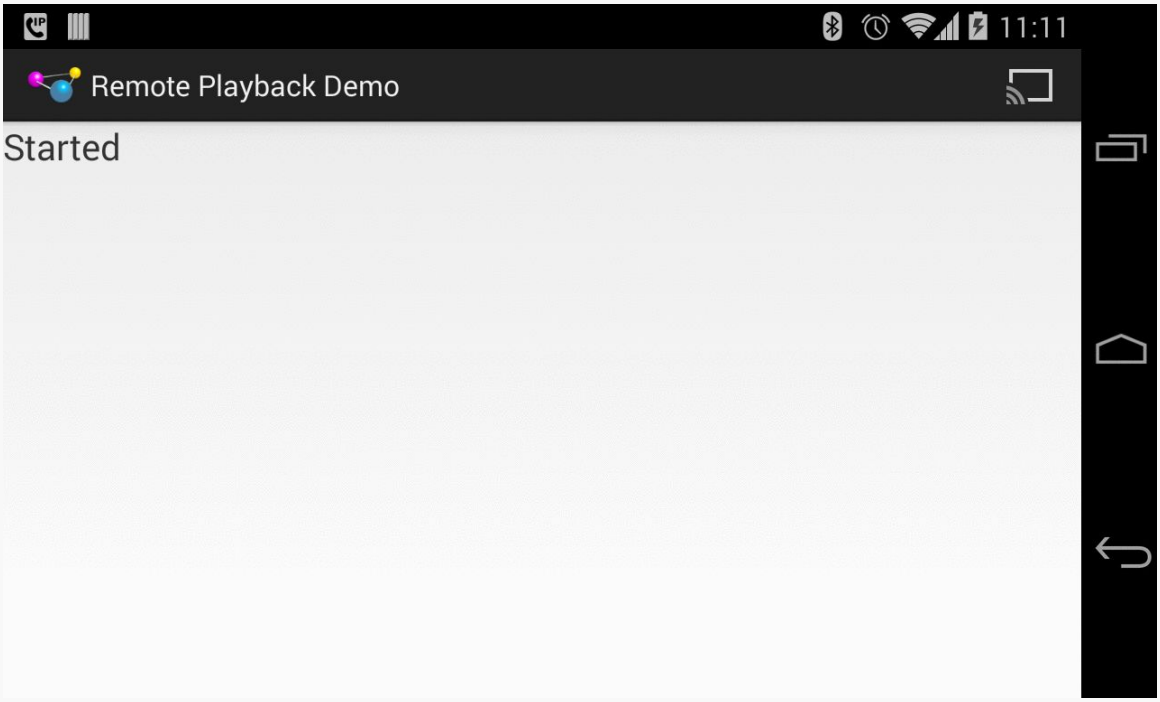


Figure 630: RemotePlaybackClient Demo, on a Nexus 4, As Initially Launched

As before, tapping on the "cast" action bar item pops up our dialog of available routes:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

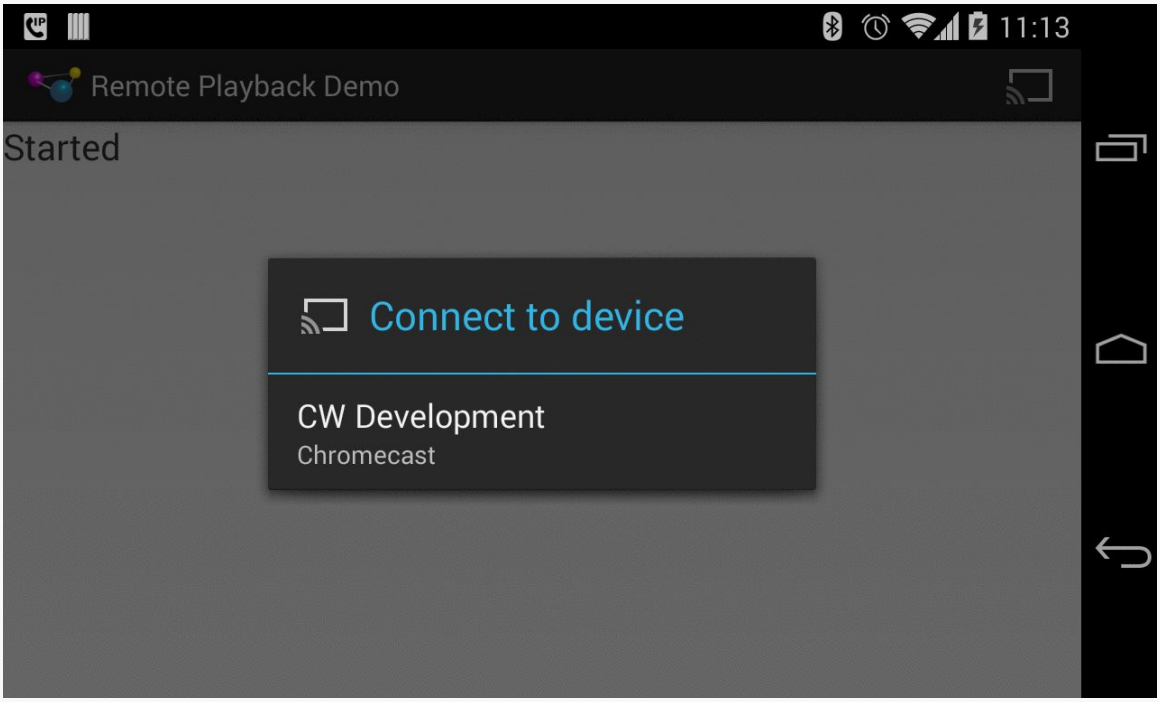


Figure 631: RemotePlaybackClient Demo, on a Nexus 4, Showing Available Routes

Connecting and Session Management

When the user selects a route, our `MediaRouter.Callback` (`cb`) is called with `onRouteSelected()`. Similarly, if the user elects to disconnect via the `MediaRouteActionProvider`, our `Callback` is called with `onRouteUnselected()`. In the `MediaRouter.Callback` implementation inside `PlaybackFragment`, those events route to `connect()` and `disconnect()` methods, respectively, after logging a message to the transcript:

```
private MediaRouter.Callback cb=new MediaRouter.Callback() {
  @Override
  public void onRouteSelected(MediaRouter router,
                              MediaRouter.RouteInfo route) {
    logToTranscript(getActivity().getString(R.string.route_selected));
    connect(route);
  }

  @Override
  public void onRouteUnselected(MediaRouter router,
                                MediaRouter.RouteInfo route) {
    logToTranscript(getActivity().getString(R.string.route_unselected));
    disconnect();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  };
```

The `connect()` method handles connecting to the remote playback device and starting a session:

```
  private void connect(MediaRouter.RouteInfo route) {
    client=
        new RemotePlaybackClient(getActivity().getApplication(), route);

    if (client.isRemotePlaybackSupported()) {
      logToTranscript(getActivity().getString(R.string.connected));

      if (client.isSessionManagementSupported()) {
        client.startSession(null, new SessionActionCallback() {
          @Override
          public void onResult(Bundle data, String sessionId,
                               MediaSessionStatus sessionStatus) {
            logToTranscript(getActivity().getString(R.string.session_started));
            updateMenu();
          }

          @Override
          public void onError(String error, int code, Bundle data) {
            logToTranscript(getActivity().getString(R.string.session_failed));
          }
        });
      }
      else {
        getActivity().supportInvalidateOptionsMenu();
      }
    }
    else {

logToTranscript(getActivity().getString(R.string.remote_playback_not_supported));
      client=null;
    }
  }
```

All of that, though, requires a bit more explanation.

What's a Session?

The objective of the `connect()` method is to establish a "session" with the `RemotePlaybackClient`. In Android's terms, a "session" is the state associated with an application's interactions with the remote playback client. In principle, the session could be shared among several instances of the app, such as several people contributing tracks to a dynamic playlist for audio playback at a party. Here, though, we are simply focused on having this one application instance have a session.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In principle, not all remote playback clients may support session management. In those cases, everybody is considered to be part of the same session. The test device for this sample (Chromecast) does support session management, however.

Connecting the Client

Connecting to the remote playback device is simply a matter of creating an instance of `RemotePlaybackClient`, specifying the route to connect to:

```
    client=
        new RemotePlaybackClient(getActivity().getApplication(), route);
```

Here, we use `getActivity().getApplication()` in the `RemotePlaybackClient` constructor. That is because we want to hold onto this `RemotePlaybackClient` instance across configuration changes, so we can easily maintain our session. Since we do not know what `RemotePlaybackClient` may hold onto given the supplied `Context`, and since we do not want to leak our activity by retaining a reference to it, we use the global `Application` instance, for a "leak-resistant" `Context`.

We also call `isRemotePlaybackSupported()` to confirm that, indeed, the `RemotePlaybackClient` is connected to something that supports remote playback. This should always return `true` in this case, as we are only interested in remote playback routes. But, a little defensive programming never hurts.

Assuming that is all OK, we log a "connected" message to the transcript and continue on to start our session.

Starting a Session

`isSessionManagementSupported()` on `RemotePlaybackClient` will indicate if the device supports explicit session management or not. If not, we will use the default implicit session and just continue on.

Otherwise, we call `startSession()` to explicitly start a session. This takes an optional `Bundle` of additional information to send in the start-session request to the device (or `null` if unused), plus a `SessionActionCallback`. The `SessionActionCallback` is supposed to be called when the session is ready for use. Surprisingly enough, this actually works... for `startSession()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `SessionActionCallback` will be called with `onResult()` for success and `onError()` for failure. In either case, we log a message to the transcript indicating the status.

In addition, if we have a session — either explicitly created via `startSession()` or implicitly created for devices without explicit session management — we call an `updateMenu()` method to update the action bar items.

About the Action Bar

The fragment maintains two `boolean` values representing key states in the operation of the playback:

1. `isPlaying` indicates if playback was started and not yet stopped
2. `isPaused` indicates if playback was paused and not yet resumed

The aforementioned `updateMenu()` implementation uses those, plus the existence of a non-`null` `client`, to configure the action bar items:

```
  private void updateMenu() {
    if (menu != null) {
      menu.findItem(R.id.stop).setVisible(client != null && isPlaying);
      menu.findItem(R.id.pause).setVisible(client != null && isPlaying
                                           && !isPaused);
      menu.findItem(R.id.play)
          .setVisible(client != null && (!isPlaying || isPaused));
    }
  }
```

Specifically:

- When we are not playing, the `play` item is visible; when we *are* playing, the `stop` item is visible
- When we are not paused, the `pause` item is visible (`play` serves "double duty", handling starting playback from a stopped state and resuming playback from a paused state)

This is based on a cached copy of the `Menu` object, saved in `onCreateOptionsMenu()` as part of setting up the action bar:

```
  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    this.menu=menu;
    inflater.inflate(R.menu.main, menu);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    updateMenu();

    MenuItem item=menu.findItem(R.id.route_provider);
    MediaRouteActionProvider provider=
        (MediaRouteActionProvider)MenuItemCompat.getActionProvider(item);

    provider.setRouteSelector(selector);
  }
```

This is also where the logic shown previously for configuring the `MediaRouteActionProvider` resides.

Session IDs

A session has a `String` identifier. In principle, this can be shared with other instances of your application, to allow for shared management of the session.

In the case of this sample, the session ID is merely logged to the transcript for all messages that are tied to an active session.

Hence, when the user chooses a remote playback route from the `MediaRouteActionProvider`, the resulting UI should resemble:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

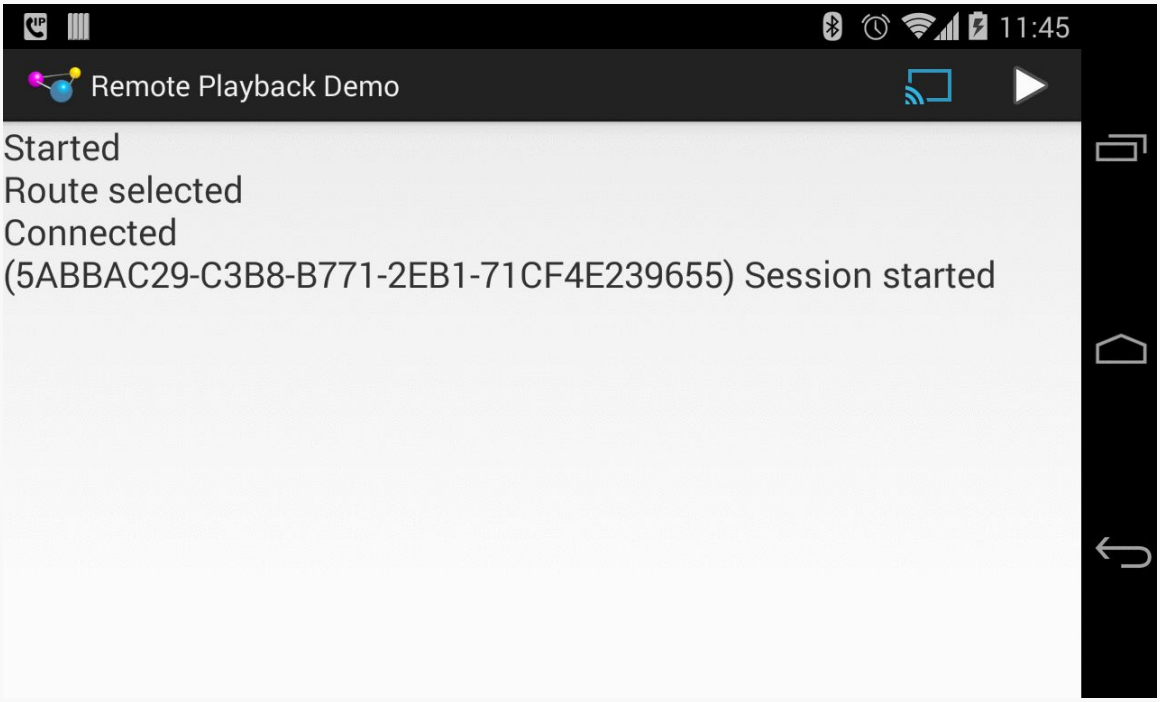


Figure 632: RemotePlaybackClient Demo, on a Nexus 4, Showing an Active Session

We see that we have connected to the client and started our session, and the `play` action bar item is now available to start playback of some media.

Playing

The `play` action bar item is tied to a `play()` method via `onOptionsItemSelected()`, if we are not paused:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.play:
      if (isPlaying && isPaused) {
        resume();
      }
      else {
        play();
      }

      return(true);

    case R.id.stop:
      stop();
      return(true);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    case R.id.pause:
      pause();
      return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

`play()`, in turn, uses the `play()` method on `RemotePlaybackClient` to play back a copy of "Elephants Dream", a Creative Commons-licensed video, hosted on CommonsWare's corner of the Amazon S3 service:

```
private void play() {
  logToTranscript(getActivity().getString(R.string.play_requested));

  ItemActionCallback playCB=new ItemActionCallback() {
    @Override
    public void onResult(Bundle data, String sessionId,
                         MediaSessionStatus sessionStatus,
                         String itemId, MediaItemStatus itemStatus) {
      logToTranscript(getActivity().getString(R.string.playing));
      isPlaying=true;
      updateMenu();
    }

    @Override
    public void onError(String error, int code, Bundle data) {
      logToTranscript(getActivity().getString(R.string.play_error)
          + error);
    }
  };

  client.play(Uri.parse("http://misc.commonsware.com/ed_hd_512kb.mp4"),
              "video/mp4", null, 0, null, playCB);
}
```

The `play()` method on `RemotePlaybackClient` takes a few parameters:

- The `Uri` of the media to be played back
- The MIME type of that media (or `null` if you do not know the MIME type)
- An optional `Bundle` of metadata about the media to be played, where the `Bundle` keys come from `MediaItemMetadata` class (or `null` if none)
- The starting offset in the media to begin playback from (use `0` to start from the beginning)
- An optional `Bundle` of additional data to pass to the device
- An instance of `ItemActionCallback` to be notified when playback has started or has failed

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`ItemActionCallback` is reminiscent of `SessionActionCallback`, in that `onResult()` will be called when playback begins and `onError()` will return when playback ends. The method signature of `onResult()` is slightly different, offering an ID and status of this particular media item.

In our case, we log a message to the transcript before requesting playback, then again on success or failure. On success, we also update `isPlaying` to be `true` and refresh the action bar.

Hence, once the user begins playback by tapping the `play` action bar item, the UI will look like this:

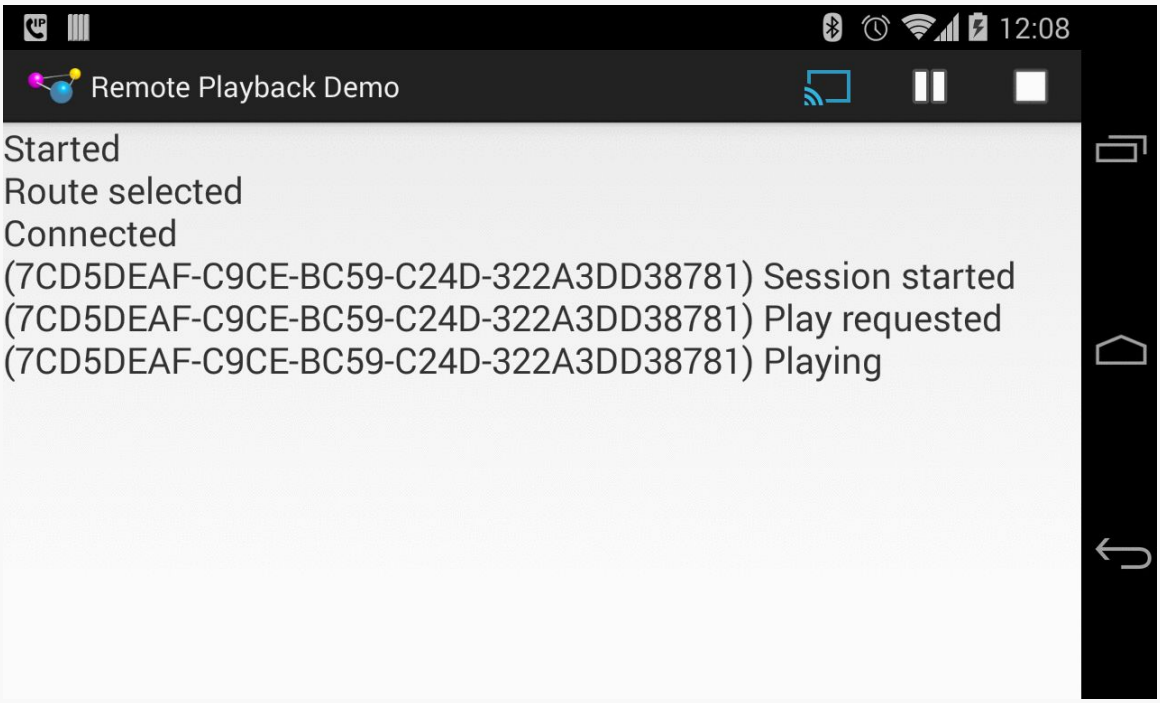


Figure 633: RemotePlaybackClient Demo, on a Nexus 4, After Playback Has Started

And, of course, the movie should be showing up on your remote playback device.

Stopping, and a Bug

The `stop()` action bar item is tied to a `stop()` method in `PlaybackFragment`. You would think that this would be very similar to starting playback — call some `stop()` method on `RemotePlaybackClient` and update the UI after playback has stopped.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

And, indeed, that is what we do... except that we have to deal with a bug:

```
  private void stop() {
    logToTranscript(getActivity().getString(R.string.stop_requested));

    StopCallback stopCB=new StopCallback();

    client.stop(null, stopCB);
    transcript.postDelayed(stopCB, 1000);
  }
```

The stop() Call, and the Bug

`stop()` on `RemotePlaybackClient` takes an optional `Bundle` (here, `null`) and a `SessionActionCallback`. The `SessionActionCallback` is supposed to be called when playback has stopped (`onResult()`) or if there was some error in processing the request (`onError()`).

In practice, neither happen when testing this on a Chromecast. This same behavior can be seen with Google's own sample code, so it would not appear to be a problem with the author's own sample.

What actually happens is that playback is indeed stopped, but the `SessionActionCallback` is not called with `onResult()` or `onError()`.

The Workaround: RunnableSessionActionCallback

Since we cannot rely upon `onResult()` to be called for us, if we have work that we need to do in that case, we have to have some sort of fallback mechanism. One crude fallback is to assume that the request succeeded if we have not received a specific response after a period of time (say, 1000 milliseconds).

To that end, this sample has `RunnableSessionActionCallback`, a `SessionActionCallback` that implements `Runnable`:

```
abstract class RunnableSessionActionCallback extends
    SessionActionCallback implements Runnable {
  abstract protected void doWork();

  private boolean hasRun=false;

  @Override
  public void onResult(Bundle data, String sessionId,
                       MediaSessionStatus sessionStatus) {
    transcript.removeCallbacks(this);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      run();
    }

    @Override
    public void run() {
      if (!hasRun) {
        hasRun=true;
        doWork();
      }
    }
  }
```

The `run()` method sees whether or not the callback has already been run from a previous `run()` call. If not, it does the work specified by the `abstract doWork()` method, to be implemented in subclasses.

`StopCallback`, as seen in the `stop()` method above, extends `RunnableSessionActionCallback` and overrides `doWork()`:

```
  private class StopCallback extends RunnableSessionActionCallback {
    @Override
    protected void doWork() {
      isPlaying=false;
      isPaused=false;
      updateMenu();
      logToTranscript(getActivity().getString(R.string.stopped));
    }
  }
```

`stop()` then not only passes the `StopCallback` to the `stop()` implementation on `RemotePlaybackClient`, but also schedules it as a `Runnable` to be invoked in 1000 milliseconds, via a call to `postDelayed()` on the `TextView` portion of the transcript. The `onResult()` implementation in `RunnableSessionActionCallback` calls `removeCallbacks()`, so we do not bother invoking the posted `Runnable` if that is not needed.

The `doWork()` implementation in `StopCallback` updates our flags, refreshes the action bar, and logs a message to the transcript. The result will look like:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

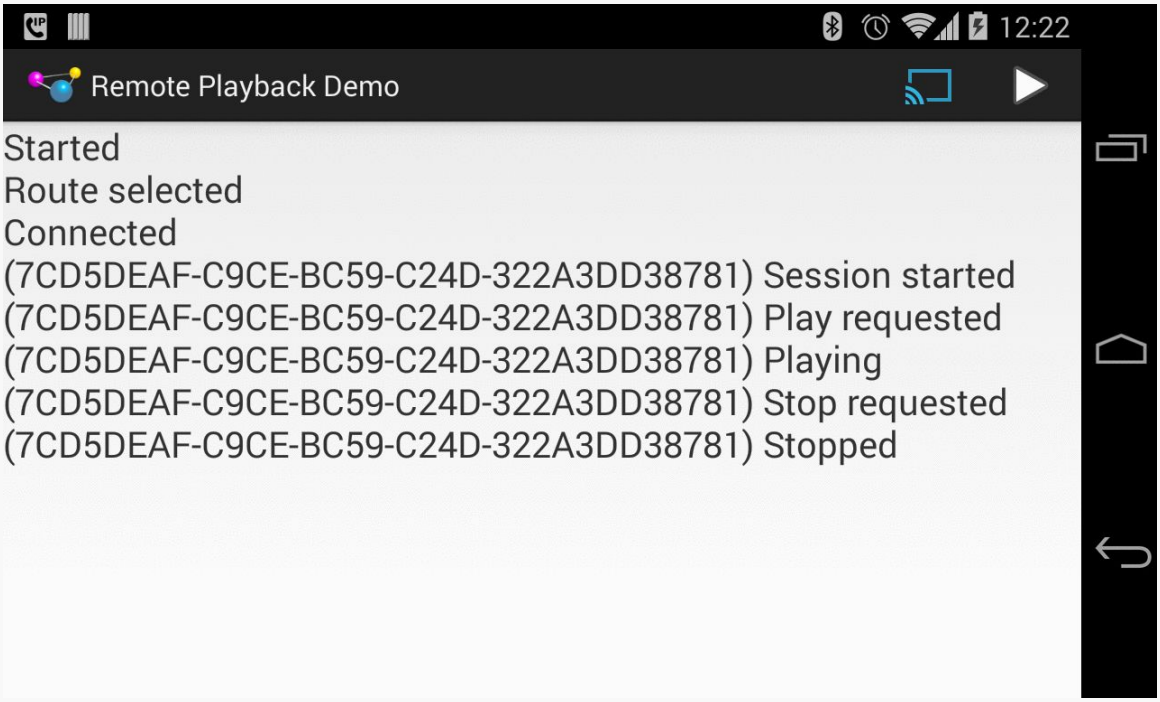


Figure 634: RemotePlaybackClient Demo, on a Nexus 4, After Playback Has Stopped

This sample also does not handle the case where the media completes playback on its own, insofar as this event is not detected, to update the action bar. This will be added in a future version of this sample, if further bugs allow such support to actually work.

Pausing and Resuming

Similarly, the `pause` action bar item forwards to a `pause()` method that calls `pause()` on the `RemotePlaybackClient`:

```
  private void pause() {
    logToTranscript(getActivity().getString(R.string.pause_requested));

    PauseCallback pauseCB=new PauseCallback();

    client.pause(null, pauseCB);
    transcript.postDelayed(pauseCB, 1000);
  }
```

That, in turn, uses `PauseCallback`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private class PauseCallback extends RunnableSessionActionCallback {
    @Override
    protected void doWork() {
      isPaused=true;
      updateMenu();
      logToTranscript(getActivity().getString(R.string.paused));
    }
  }
```

This updates the action bar and logs messages to the transcript, similar to the `stop()` behavior. It also should successfully pause playback on the remote device.

The `play` action bar item routes to `resume()` if playback is paused:

```
  private void resume() {
    logToTranscript(getActivity().getString(R.string.resume_requested));

    ResumeCallback resumeCB=new ResumeCallback();

    client.resume(null, resumeCB);
    transcript.postDelayed(resumeCB, 1000);
  }
```

That, in turn, uses `ResumeCallback`:

```
  private class ResumeCallback extends RunnableSessionActionCallback {
    @Override
    protected void doWork() {
      isPaused=false;
      updateMenu();
      logToTranscript(getActivity().getString(R.string.resumed));
    }
  }
```

This too updates the action bar and logs messages to the transcript, in addition to resuming playback on the remote device.

Disconnecting

A call to `disconnect()` on `PlaybackFragment` is triggered from two locations:

- `onRouteUnselected()` in our `MediaRouter.Callback`, such as when the user uses the `MediaRouteActionProvider` to disconnect from the route
- `onDestroy()`, as part of general cleanup of the fragment

`disconnect()` should reverse the work done in `connect()`, ending our session and releasing the client:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private void disconnect() {
    isPlaying=false;
    isPaused=false;

    if (client != null) {
      logToTranscript(getActivity().getString(R.string.session_ending));
      EndSessionCallback endCB=new EndSessionCallback();

      if (client.isSessionManagementSupported()) {
        client.endSession(null, endCB);
      }

      transcript.postDelayed(endCB, 1000);
    }
  }
```

This simply calls the `endSession()` method on the `RemotePlaybackClient`, supplying an `EndSessionCallback` to be notified (theoretically) of when the session has been torn down. But it only calls `endSession()` if session management is supported; otherwise, we would get a runtime error.

To be sure we complete the disconnection, though, we schedule the `EndSessionCallback` as seen in the `stop()`, `pause()`, and `resume()` methods. `EndSessionCallback` calls `release()` on the `RemotePlaybackClient`, to indicate that we are done with it, before setting `client` to `null`, refreshing the action bar, and logging something to the transcript:

```
  private class EndSessionCallback extends
      RunnableSessionActionCallback {
    @Override
    protected void doWork() {
      client.release();
      client=null;

      if (getActivity() != null) {
        updateMenu();
        logToTranscript(getActivity().getString(R.string.session_ended));
      }
    }
  }
```

Other Remote Playback Features

There are other things that `RemotePlaybackClient` offers that are not shown in this sample:

- `enqueue()` allows you to build up a queue of media to be played back in the current session. This could be used by an individual or, in principle, by

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

several people using the same app with a shared session ID. `remove()` allows you to remove specific items from the playback queue. These methods only work if `isQueueingSupported()` returns `true`.

- `getStatus()` will return information about the currently-playing piece of media, while `getSessionStatus()` will return information about the overall session. You can also find out about these changes on the fly by registering with `setStatusCallback()`.
- `seek()` allows you to move the playback to a new offset within the media, for "rewind" and "fast-forward" functionality. The status APIs (above) can tell you where you are in the playback, so you can determine the appropriate offset to seek to.

The CWAC-MediaRouter Version

A version of this sample project, using the CWAC-MediaRouter edition of `MediaRouteActionProvider`, can be found in the `demo-remoteplayback/` directory of the CWAC-MediaRouter repository.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Supporting External Displays

Android 4.2 inaugurated support for applications to control what appears on an external or "secondary" display (e.g., TV connected via HDMI), replacing the default screen mirroring. This is largely handled through a `Presentation` object, where you declare the UI that goes onto the external display, in parallel with whatever your activity might be displaying on the primary screen.

In this chapter, we will review how Android supports these external displays, how you can find out if an external display is attached, and how you can use `Presentation` objects to control what is shown on that external display.

The author would like to thank Mark Allison, whose "Multiple Screens" blog post series helped to blaze the trail for everyone in this space.

Prerequisites

In addition to the core chapters, you should read the chapter on dialogs and the chapter on `MediaRouter` before reading this chapter.

A History of External Displays

In this chapter, "external displays" refers to a screen that is temporarily associated with an Android device, in contrast with a "primary screen" that is where the Android device normally presents its user interface. So, most Android devices connected to a television via HDMI would consider the television to be a "external display", with the touchscreen of the device itself as the "primary screen". However, a Android TV box or OUYA console connected to a television via HDMI would consider the television to be the "primary screen", simply because there is no other

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

screen. Some devices themselves may have multiple screens, such as the Sony Tablet P — what those devices do with those screens will be up to the device.

Historically, support for external displays was manufacturer-dependent. Early Android devices had no ability to be displayed on an external display except through so-called "software projectors" like Jens Riboe's Droid@Screen. Some Android 2.x devices had ports that allowed for HDMI or composite connections to a television or projector. However, control for what would be displayed resided purely in the hands of the manufacturer. Some manufacturers would display whatever was on the touchscreen (a.k.a., "mirroring"). Some manufacturers would do that, but only for select apps, like a built-in video player.

Android 3.0 marked the beginning of Android's formal support for external displays, as the Motorola XOOM supported mirroring of the LCD's display via an micro-HDMI port. This mirroring was supplied by the core OS, not via device-dependent means. Any Android 3.0+ device with some sort of HDMI connection (e.g., micro-HDMI port) should support this same sort of mirroring capability.

However, mirroring was all that was possible. There was no means for an application to have something on the external display (e.g., a video) and something *else* on the primary screen (e.g., playback controls plus IMDB content about the movie being watched).

Android 4.2 changed that, with the introduction of `Presentation`.

What is a Presentation?

A `Presentation` is a container for displaying a UI, in the form of a `View` hierarchy (like that of an activity), on an external display.

You can think of a `Presentation` as being a bit like a `Dialog` in that regard. Just as a `Dialog` shows its UI separate from its associated activity, so does a `Presentation`. In fact, as it turns out, `Presentation` *inherits from* `Dialog`.

The biggest difference between a `Presentation` and an ordinary `Dialog`, of course, is where the UI is displayed. A `Presentation` displays on an external display; a `Dialog` displays on the primary screen, overlaying the activity. However, this difference has a profound implication: the *characteristics* of the external display, in terms of size and density, are likely to be different than those of a primary screen.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Hence, the resources used by the UI on an external display may be different than the resources used by the primary screen. As a result, **the `Context` of the `Presentation` is not the `Activity`**. Rather, it is a separate `Context`, one whose `Resources` object will use the proper resources based upon the external display characteristics.

This seemingly minor bit of bookkeeping has some rippling effects on setting up your `Presentation`, as we will see as this chapter unfolds.

Playing with External Displays

To write an app that uses an external display via a `Presentation`, you will need Android 4.2 or higher.

Beyond that, though, you will also need an external display of some form. Presently, you have three major options: emulate it, use a screen connected via some sort of cable, or use Miracast for wireless external displays.

Emulated

Even without an actual external display, you can lightly test your `Presentation`-enabled app via the Developer Options area of Settings on your Android 4.2 device. There, in the Drawing category, you will see the "Simulate secondary displays" preference:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

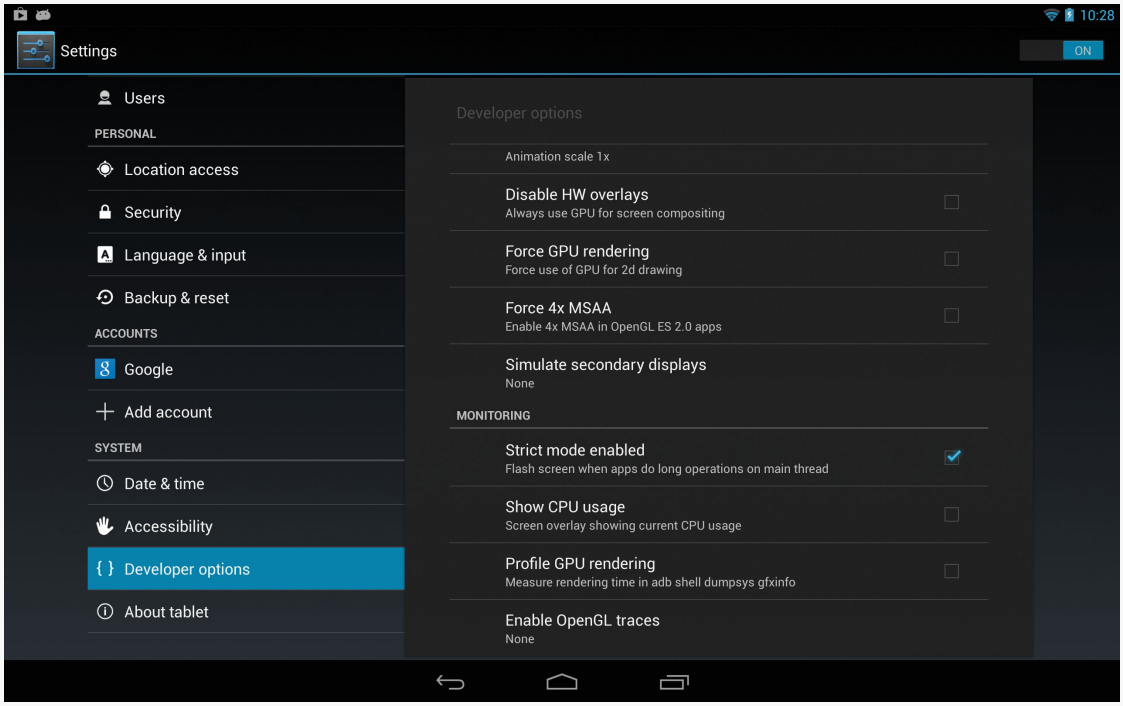


Figure 635: Nexus 10 "Simulate secondary displays" Preference

Tapping that will give you various options for what secondary display to emulate:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

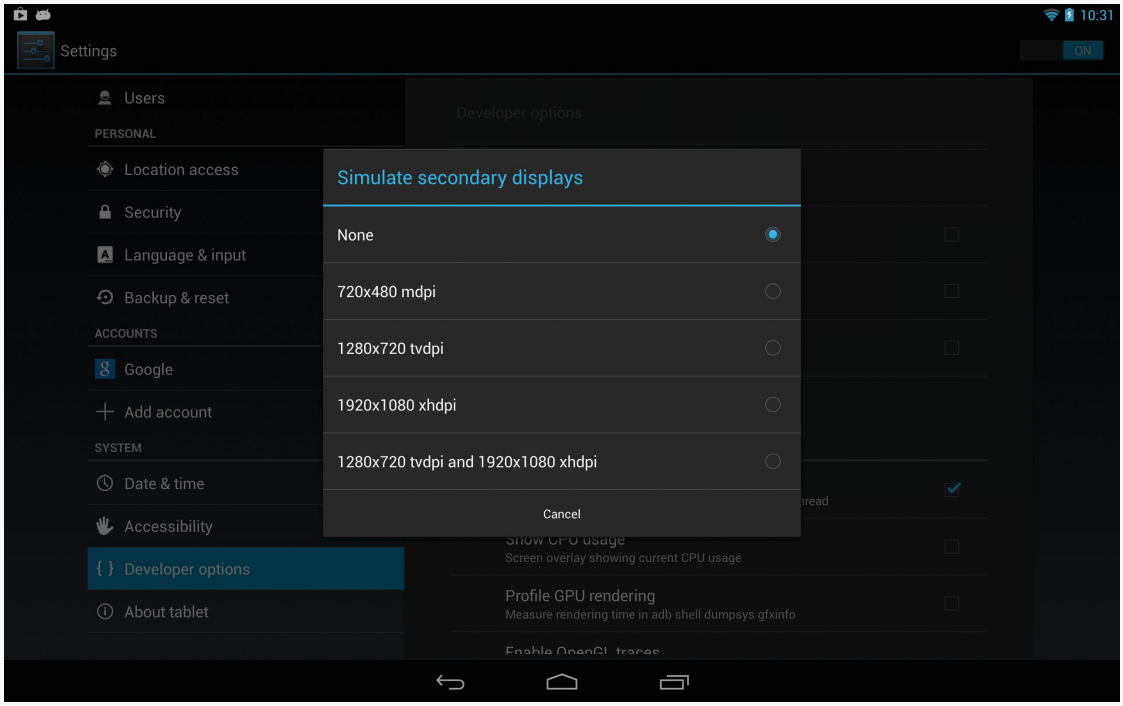


Figure 636: Nexus 10 "Simulate secondary displays" Options

Tapping one of those will give you a small window in the upper-left corner, showing the contents of the external display, overlaid on top of your regular screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

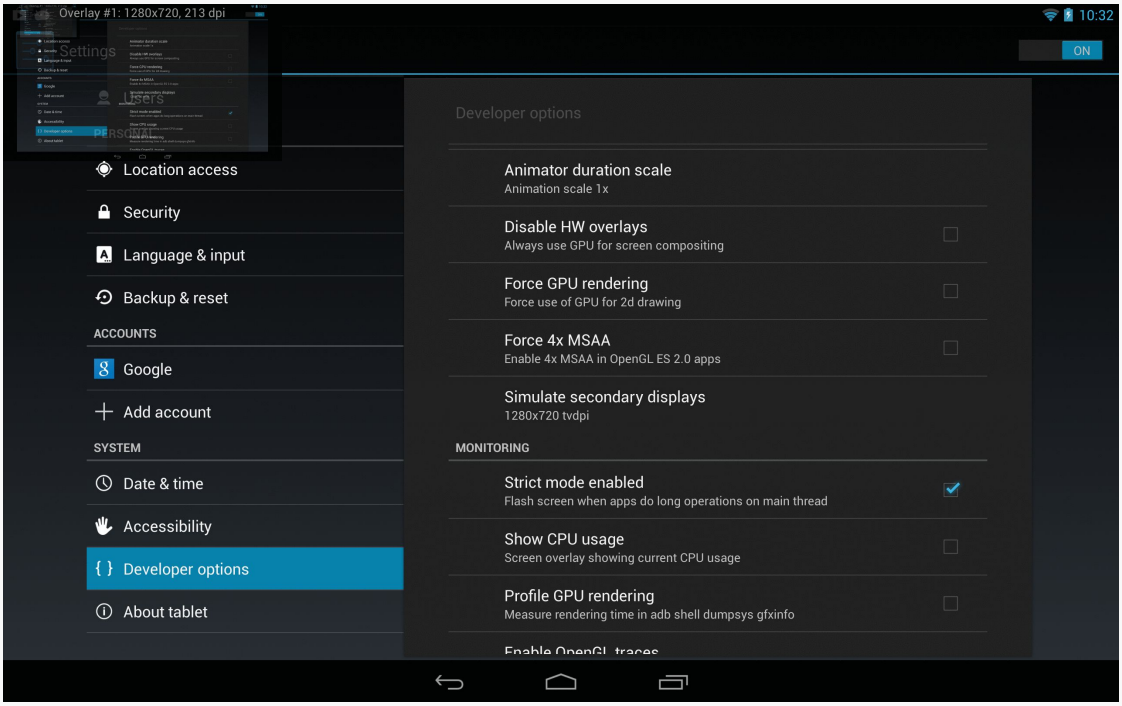


Figure 637: Nexus 10, Simulating a 720p external display

Normally, that will show a mirrored version of the primary screen, but with a `Presentation`-enabled app, it will show what is theoretically shown on the real external display.

However, there are limits with this technology:

- You will see this option on an Android emulator, but it may not work, particularly if you are not capable of using the "Host GPU Support" option. At the time of this writing, it works on the x86 Android 4.2 emulator image, but not the x86 Android 4.3 or 4.4 emulator image, and the ARM emulators are likely to be far too slow.
- The external display is rather tiny, making it difficult for you to accurately determine if everything is sized appropriately.
- The external display occludes part of the screen, overlaying your activities, though you can at least drag it around the screen to move it out of your way as needed.

In practice, before you ship a `Presentation`-capable app, you will want to test it with an actual physical external display.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

HDMI

If you have a device with HDMI-out capability, and you have the appropriate cable, you can simply plug that cable between your device and the display. "Tuning" the display to use that specific HDMI input port should cause your device's screen contents to be mirrored to that display. Once this is working, you should be able to control the contents of that display using `Presentation`.

MHL

Mobile High-Definition Link, or MHL for short, is a relatively new option for connections to displays. On many modern Android devices, the micro USB port supports MHL as well. Some external displays have MHL ports, in which case a male-to-male MHL direct cable will connect the device to the display. Otherwise, MHL can be converted to HDMI via adapters, so an MHL-capable device can attach to any HDMI-compliant display.

SlimPort

SlimPort is another take on the overload-the-micro-USB-port-for-video approach. MHL is used on substantially more devices, but SlimPort appears on several of the Nexus-series devices (Nexus 4, Nexus 5, and the 2013 generation of the Nexus 7). Hence, while users will be more likely to have an MHL device, *developers* may be somewhat more likely to have a SlimPort device, given the popularity of Nexus devices among Android app developers.

From the standpoint of your programming work, MHL and SlimPort are largely equivalent — there is nothing that you need to do with your `Presentation` to address either of those protocols, let alone anything else like native HDMI.

Miracast

There are a few wireless display standards available. Android 4.2 supports Miracast, based upon WiFiDirect. This is also supported by some devices running earlier versions of Android, such as some Samsung devices (where Miracast is sometimes referred to as "AllShare Cast"). However, unless and until those devices get upgraded to Android 4.2, you cannot control what they display, except perhaps through some manufacturer-specific APIs.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

On a Miracast-capable device, going into Settings > Displays > Wireless display will give you the ability to toggle on wireless display support and scan for available displays:

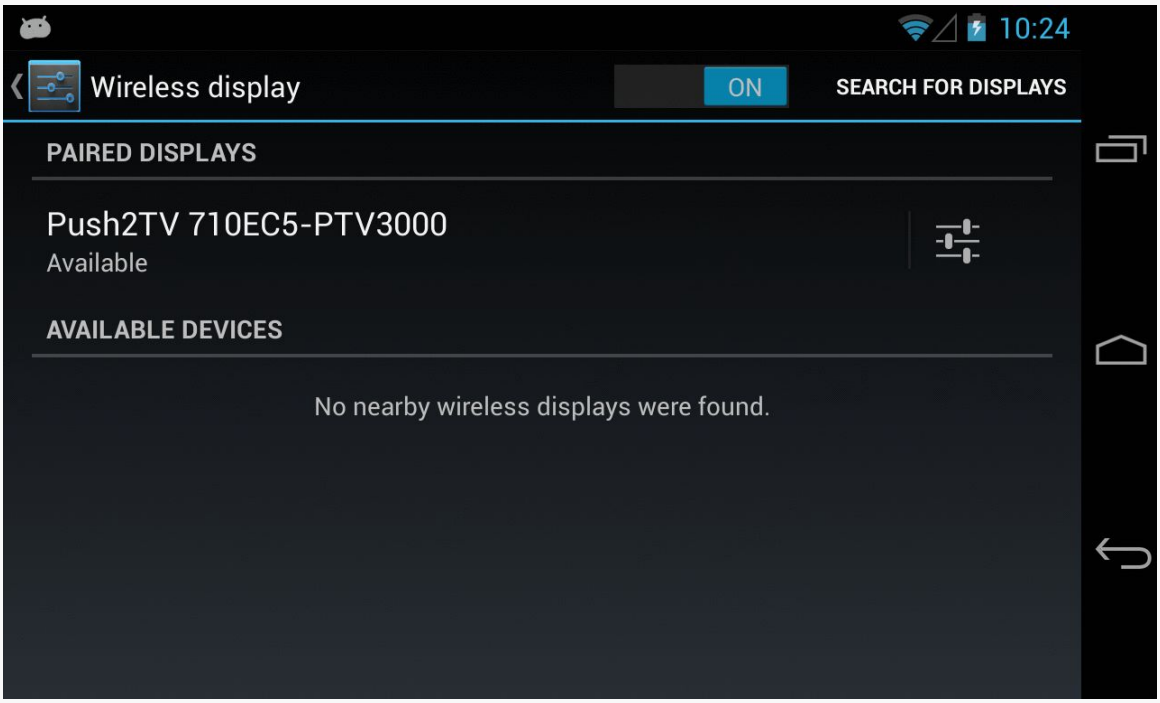


Figure 638: Nexus 4 Wireless Display Settings

You can then elect to attach to one of the available wireless displays and get your screen mirrored, and later use this with your `Presentation`-enabled app.

Of course, you also need some sort of Miracast-capable display. As of early 2013, there were few of these. However, you can also get add-on boxes that connect to normal displays via HDMI and make them available via Miracast. One such box is the Netgear PTV3000, whose current firmware supports Miracast along with other wireless display protocols.

Note that Miracast uses a compressed protocol, to minimize the bandwidth needed to transmit the video. This, in turn, can cause some lag.

Note that Intel's WiDi is an extended version of Miracast.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

WirelessHD

An up-and-coming competitor to Miracast is WirelessHD. WirelessHD has greater bandwidth requirements. On the other hand, it avoids compression, and therefore the lag that you experience with Miracast. At the time of this writing, though, no WirelessHD-native Android devices are available.

Detecting Displays

Of course, we can only present a `Presentation` on an external display if there is, indeed, such a screen available. There are two approaches for doing this: using `DisplayManager` and using `MediaRouter`. We examined `MediaRouter` for detecting live video routes in a preceding chapter, so let's focus here on `DisplayManager`.

`DisplayManager` is a system service, obtained by calling `getSystemService()` and asking for the `DISPLAY_SERVICE`.

Once you have a `DisplayManager`, you can ask it to give you a list of all available displays (`getDisplays()` with zero arguments) or all available displays in a certain category (`getDisplays()` with a single `String` parameter). As of API Level 17, the only available display category is `DISPLAY_CATEGORY_PRESENTATION`. The difference between the two flavors of `getDisplays()` is just the sort order:

- The zero-argument `getDisplays()` returns the `Display` array in arbitrary order
- The one-argument `getDisplays()` will put the `Display` objects matching the identified category earlier in the array

These would be useful if you wanted to pop up a list of available displays to ask the user which `Display` to use.

You can also register a `DisplayManager.DisplayListener` with the `DisplayManager` via `registerDisplayListener()`. This listener will be called when displays are added (e.g., HDMI cable was connected), removed (e.g., HDMI cable was disconnected), or changed. It is not completely clear what would trigger a "changed" call, though possibly an orientation-aware display might report back the revised height and width.

Note that while `DisplayManager` was added in API Level 17, `Display` itself has been around since API Level 1, though some additions have been made in more recent

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android releases. But, this may mean that you can pass the `Display` object around to code supporting older devices without needing to constantly check for SDK level or add the `@TargetApi()` annotation.

A Simple Presentation

Let's take a look at a small sample app that demonstrates how we can display custom content on an external display using a `Presentation`. The app in question can be found in the Presentation/Simple sample project.

The Presentation Itself

Since `Presentation` extends from `Dialog`, we provide the UI to be displayed on the external display via a call to `setContentView()`, much like we would do in an activity. Here, we just create a `WebView` widget in Java, point it to some Web page, and use it:

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
private class SimplePresentation extends Presentation {
  SimplePresentation(Context ctxt, Display display) {
    super(ctxt, display);
  }

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    WebView wv=new WebView(getContext());

    wv.loadUrl("http://commonsware.com");

    setContentView(wv);
  }
}
```

However, there are two distinctive elements of our implementation:

- Our constructor takes a `Context` (typically the `Activity`), along with a `Display` object indicating where the UI should be presented.
- Our call to the `WebView` constructor uses `getContext()`, instead of the `Activity` object. In this case, that may have no real-world effect, as `WebView` is not going to be using any of our resources. But, had we used a `LayoutInflater` for inflating our UI, we would need to use one created from `getContext()`, not from the activity itself.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Detecting the Displays

We need to determine whether there is a suitable external display when our activity comes into the foreground. We also need to determine if an external display was added or removed while we are in the foreground.

So, in `onResume()`, if we are on an Android 4.2 or higher device, we will get connected to the `MediaRouter` to handle those chores:

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
@Override
protected void onResume() {
  super.onResume();

  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
    if (cb==null) {
      cb=new RouteCallback();
      router=(MediaRouter)getSystemService(MEDIA_ROUTER_SERVICE);
    }

    handleRoute(router.getSelectedRoute(MediaRouter.ROUTE_TYPE_LIVE_VIDEO));
    router.addCallback(MediaRouter.ROUTE_TYPE_LIVE_VIDEO, cb);
  }
}
```

Specifically, we:

- Create an instance of `RouteCallback`, an inner class of our activity that extends `SimpleCallback`
- Use `getSystemService()` to obtain a `MediaRouter`
- Call a `handleRoute()` method on our activity that will update our UI based upon the current video route, obtained by calling `getSelectedRoute()` on the `MediaRouter`
- Register the `RouteCallback` object with the `MediaRouter` via `addCallback()`

The `RouteCallback` object simply overrides `onRoutePresentationDisplayChanged()`, which will be called whenever there is a change in what screens are available and considered to be the preferred modes for video. There, we just call that same `handleRoute()` method that we called in `onResume()`:

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN)
private class RouteCallback extends SimpleCallback {
  @Override
  public void onRoutePresentationDisplayChanged(MediaRouter router,
                                                RouteInfo route) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      handleRoute(route);
    }
  }
```

Hence, our business logic for showing the presentation is isolated in one method, `handleRoute()`.

Our `onPause()` method will undo some of the work done by `onResume()`, notably removing our `RouteCallback`. We will examine that more closely in the next section.

Showing and Hiding the Presentation

Our `handleRoute()` method will be called with one of two parameter values:

- The `RouteInfo` of the active route we should use for displaying the `Presentation`
- `null`, indicating that there is no route for such content, other than the primary screen

If we are passed the `RouteInfo`, it may represent the route we are already using, or possibly it may represent a different route entirely.

We need to handle all of those cases, even if some (switching directly from one route to another) may not necessarily be readily testable.

Hence, our `handleRoute()` method does its best:

```
  @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
  private void handleRoute(RouteInfo route) {
    if (route == null) {
      clearPreso();
    }
    else {
      Display display=route.getPresentationDisplay();

      if (route.isEnabled() && display != null) {
        if (preso == null) {
          showPreso(route);
          Log.d(getClass().getSimpleName(), "enabled route");
        }
        else if (preso.getDisplay().getDisplayId() != display.getDisplayId()) {
          clearPreso();
          showPreso(route);
          Log.d(getClass().getSimpleName(), "switched route");
        }
        else {
          // no-op: should already be set
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        }
      }
      else {
        clearPreso();
        Log.d(getClass().getSimpleName(), "disabled route");
      }
    }
  }
```

There are five possibilities handled by this method:

- If the route is `null`, then we should no longer be displaying the `Presentation`, so we call a `clearPreso()` method that will handle that
- If the route exists, but is disabled or is not giving us a `Display` object, we also assume that we should no longer be displaying the `Presentation`, so we call `clearPreso()`
- If the route exists and seems ready for use, and we are not already showing a `Presentation` (our `preso` data member is `null`), we need to show the `Presentation`, which we delegate to a `showPreso()` method
- If the route exists, seems ready for use, but we are already showing a `Presentation`, and the ID of the new `Display` is different than the ID of the `Display` our `Presentation` had been using, we use *both* `clearPreso()` and `showPreso()` to switch our `Presentation` to the new `Display`
- If the route exists, seems ready for use, but we are already showing a `Presentation` on this `Display`, we do nothing and wonder why `handleRoute()` got called

Showing the `Presentation` is merely a matter of creating an instance of our `SimplePresentation` and calling `show()` on it, like we would a regular `Dialog`:

```
  @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
  private void showPreso(RouteInfo route) {
    preso=new SimplePresentation(this, route.getPresentationDisplay());
    preso.show();
  }
```

Clearing the `Presentation` calls `dismiss()` on the `Presentation`, then sets the `preso` data member to `null` to indicate that we are not showing a `Presentation`:

```
  @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
  private void clearPreso() {
    if (preso != null) {
      preso.dismiss();
      preso=null;
    }
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Our `onPause()` uses `clearPreso()` and `removeCallback()` to unwind everything:

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
@Override
protected void onPause() {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
    clearPreso();

    if (router != null) {
      router.removeCallback(cb);
    }
  }

  super.onPause();
}
```

The Results

If you run this with no external display, you will just see a plain `TextView` that is the UI for our primary screen:

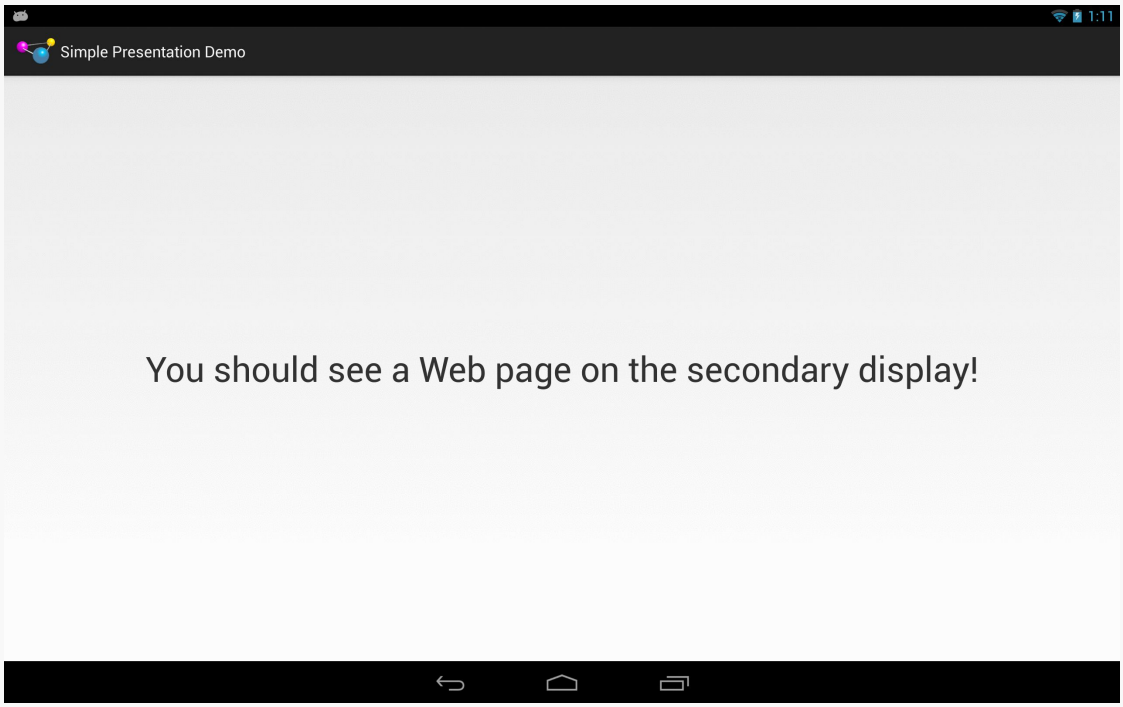


Figure 639: Nexus 10, No Emulated Secondary Display, Showing Sample App

If you run this *with* an external display, the external display will show our `WebView`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

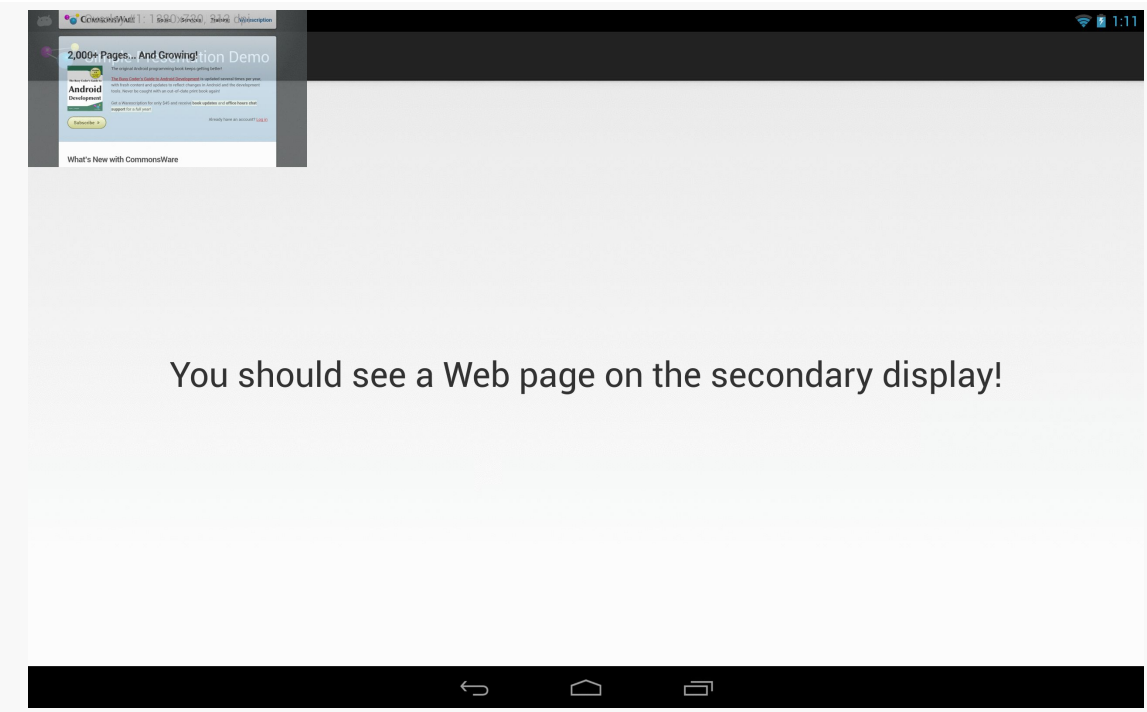


Figure 640: Nexus 10, With Emulated Secondary Display, Showing Sample App

A Simpler Presentation

There was a fair bit of code in the previous sample for messing around with `MediaRouter` and finding out about changes in the available displays.

To help simplify apps using `Presentation`, the author of this book maintains a library, `CWAC-Presentation`, with various reusable bits of code for managing `Presentations`.

One piece of this is `PresentationHelper`, which isolates all of the display management logic in a single reusable object. In this section, we will examine how to use `PresentationHelper`, then how `PresentationHelper` itself works, using `DisplayManager` under the covers.

Getting a Little Help

Our Presentation/Simpler sample project has the `CWAC-Presentation` JAR in its `libs/` directory, giving us access to `PresentationHelper`. Our `MainActivity` in the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

sample creates an instance of `PresentationHelper` in `onCreate()`, stashing the object in a data member:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  setContentView(R.layout.activity_main);
  helper=new PresentationHelper(this, this);
}
```

The constructor for `PresentationHelper` takes two parameters:

- a `Context` object, one that should be valid for the life of the helper, typically the `Activity` that creates the helper, and
- a implementation of `PresentationHelper.Listener` — in this case, the interface is implemented on `MainActivity` itself

The activity that creates the helper must forward `onPause()` and `onResume()` lifecycle methods to the equivalent methods on the helper:

```
@Override
public void onResume() {
  super.onResume();
  helper.onResume();
}

@Override
public void onPause() {
  helper.onPause();
  super.onPause();
}
```

The implementer of `PresentationHelper.Listener` also needs to have `showPreso()` and `clearPreso()` methods, much like the ones from the original `Presentation` sample in this chapter. `showPreso()` will be passed a `Display` object and should arrange to display a `Presentation` on that `Display`:

```
@Override
public void showPreso(Display display) {
  preso=new SimplerPresentation(this, display);
  preso.show();
}
```

`clearPreso()` should get rid of any outstanding `Presentation`. It is passed a `boolean` value, which will be `true` if we simply lost the `Display` we were using (and so the activity might want to display the `Presentation` contents elsewhere, such as in the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

activity itself), or `false` if the activity is moving to the background (triggered via `onPause()`):

```
  @Override
  public void clearPreso(boolean showInline) {
    if (preso != null) {
      preso.dismiss();
      preso=null;
    }
  }
```

The implementations here are pretty much the same as the ones used in the previous example. `PresentationHelper` has handled all of the `Display`-management events – our activity can simply focus on showing or hiding the `Presentation` on demand.

Help When You Need It

In many respects, the `PresentationHelper` from the CWAC-Presentation project works a lot like the logic in the original `Presentation` sample's `MainActivity`, detecting various states and calling `showPreso()` and `clearPreso()` accordingly. However, `PresentationHelper` uses a different mechanism for this — `DisplayManager`.

The `PresentationHelper` constructor just stashes the parameters it is passed in data members and obtains a `DisplayManager` via `getSystemService()`, putting it in another data member:

```
  public PresentationHelper(Context ctxt, Listener listener) {
    this.listener=listener;

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
      mgr=
          (DisplayManager)ctxt.getSystemService(Context.DISPLAY_SERVICE);
    }
  }
```

`onResume()` calls out to a private `handlePreso()` method to initialize our state, and tells the `DisplayManager` to let it know as displays are attached and detached from the device, by means of `registerDisplayListener()`:

```
  public void onResume() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
      handleRoute();
      mgr.registerDisplayListener(this, null);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }
```

The `PresentationHelper` itself implements the `DisplayListener` interface, which requires three callback methods:

- `onDisplayAdded()` is called when a new output display is available
- `onDisplayChanged()` is called when an existing attached display changes its characteristics
- `onDisplayRemoved()` is called whenever a previously-attached output display has been detached

In our case, all three methods route to the same `handleRoute()` method, to update our state:

```
  @Override
  public void onDisplayAdded(int displayId) {
    handleRoute();
  }

  @Override
  public void onDisplayChanged(int displayId) {
    handleRoute();
  }

  @Override
  public void onDisplayRemoved(int displayId) {
    handleRoute();
  }
```

`handleRoute()` is where the bulk of the "business logic" of `PresentationHelper` resides:

```
  private void handleRoute() {
    if (isEnabled()) {
      Display[] displays=
          mgr.getDisplays(DisplayManager.DISPLAY_CATEGORY_PRESENTATION);

      if (displays.length == 0) {
        if (current != null || isFirstRun) {
          listener.clearPreso(true);
          current=null;
        }
      }
      else {
        Display display=displays[0];

        if (display != null && display.isValid()) {
          if (current == null) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
            listener.showPreso(display);
            current=display;
          }
          else if (current.getDisplayId() != display.getDisplayId()) {
            listener.clearPreso(true);
            listener.showPreso(display);
            current=display;
          }
          else {
            // no-op: should already be set
          }
        }
        else if (current != null) {
          listener.clearPreso(true);
          current=null;
        }
      }

      isFirstRun=false;
    }
  }
```

We get the list of attached displays from the `DisplayManager` by calling `getDisplays()`. By passing in `DISPLAY_CATEGORY_PRESENTATION`, we are asking for returned array of `Display` objects to be ordered such that the preferred display for presentations is the first element.

If the array is empty, and we already had a `current` `Display` from before (or if this is the first time `handlePreso()` has run), we call `clearPreso()` to inform the listener that there is no `Display` for presentation purposes.

If we do have a valid `Display`:

- If we were not displaying anything before, we call `showPreso()` to inform the listener to start displaying things, plus keep track of the `current` `Display` in a data member
- If we were displaying something before, but now the preferred `Display` for a `Presentation` is different (the ID value of the `Display` objects differ), we call `clearPreso()` and `showPreso()` to get the listener to switch to the new `Display`
- Otherwise, this was a spurious call to `handlePreso()`, so we do not do anything of note

If, for whatever reason, the best `Display` is not valid, we do the same thing as if we had no `Display` at all: call `clearPreso()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Finally, in `onPause()`, we call `clearPreso()` to ensure that we are no longer attempting to display anything, plus call `unregisterDisplayListener()` so we are no longer informed about changes to the mix of `Display` objects that might be available:

```
public void onPause() {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
    listener.clearPreso(false);
    current=null;

    mgr.unregisterDisplayListener(this);
  }
}
```

Presentations and Configuration Changes

One headache when using `Presentation` comes from the fact that it is a `Dialog`, which is owned by an `Activity`. If the device undergoes a configuration change, the activity will be destroyed and recreated by default, forcing you to destroy and recreate your `Dialog`. This, in turn, causes flicker on the external display, as the display briefly reverts to mirroring while this goes on.

Devices that support external displays may be orientation-locked to landscape when an external display is attached (e.g., an HDMI cable is plugged in). This reduces the odds of a configuration change considerably, as the #1 configuration change is an orientation change. However, that is not a guaranteed "feature" of Android external display support, and there are other configuration changes that could go on (e.g., devices gets plugged into a keyboard dock).

You can either just live with the flicker, or use `android:configChanges` to try to avoid the destroy/re-create cycle for the configuration change. As was noted back in the chapter on configuration changes, this is a risky approach, as it requires you to remember all your resources that might change on the configuration change and reset them to reflect the configuration change.

A "middle ground" approach is to ensure that your activity running the `Presentation` is orientation-locked to landscape mode, by adding `android:orientation="landscape"` to your `<activity>` in the manifest, then use `android:configChanges` to handle the configuration changes related to orientation:

- `orientation`
- `keyboardHidden`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- screenSize
- screenLayout

For those configuration changes, nothing should be needed to be modified in your activity, since you want to be displaying in landscape all of the time, and so you will not need to modify your use of resources. This leaves open the possibility of other configuration changes that would cause flicker on the external display, but those are relatively unlikely to occur while your activity is in the foreground, and so it may not be worth trying to address the flicker in all those cases.

Presentations as Fragments

Curiously, the support for `Presentation` is focused on `View`. There is nothing built into Android 4.2 that ties a `Presentation` to a `Fragment`. However, this can be a useful technique, one we can roll ourselves... with a bit of difficulty.

The Reuse Reality

There will be a few apps that will only want to deliver content if there is a external display on which to deliver it. However, the vast majority of apps supporting external displays will do so optionally, still supporting regular Android devices with only primary screens.

In this case, though, we have a problem: we need to show that UI *somewhere* if there is no external display to show it on. Our only likely answer is to have it be part of our primary UI.

Fragments would seem to be tailor-made for this. We could "throw" a fragment to the external display if it exists, or incorporate it into our main UI (e.g., as another page in a `ViewPager`) if the external display does not exist, or even have it be shown by some separate activity on smaller-screen devices like phones. Our business logic will already have been partitioned between the fragments — it is merely a question of where the fragment shows up.

Presentations as Dialogs

The nice thing is that `Presentation` extends `Dialog`. We already have a `DialogFragment` as part of Android that knows how to display a `Dialog` populated by a `Fragment` implementation of `onCreateView()`. `DialogFragment` even knows how to handle either being part of the main UI *or* as a separate dialog.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Hence, one could imagine a `PresentationFragment` that extends `DialogFragment` and adds the ability to either be part of the main UI on the primary screen *or* shown on an external display, should one be available.

And, in truth, it is possible to create such a `PresentationFragment`, though there are some limitations.

The Context Conundrum

The biggest limitation comes back to the `Context` used for our UI. Normally, there is only one `Context` of relevance: the `Activity`. In the case of `Presentation`, though, there is a separate `Context` that is tied to the display characteristics of the external display.

This means that `PresentationFragment` must manipulate *two* `Context` values:

- The `Activity`, if the fragment should be part of our main UI
- Some other `Context` supplied by the `Presentation`, if the fragment should be displayed in the `Presentation` on the external display

This makes creating a `PresentationFragment` class a bit tricky... though not impossible. After all, if it *were* impossible, these past several paragraphs would not be very useful.

A PresentationFragment (and Subclasses)

The Presentation/Fragment sample project has the same UI as the `Presentation/Simple` project, if there is an external display. If there is only the primary screen, though, we will elect to display the `WebView` side-by-side with our `TextView` in the main UI of our activity. And, to pull this off, we will create a `PresentationFragment` based on `DialogFragment`.

Note that this sample project has its `android:minSdkVersion` set to 17, mostly to cut down on all of the "only do this if we are on API Level 17" checks and `@TargetApi()` annotations. Getting this code to work on earlier versions of Android is left as an exercise for the reader.

In a simple `DialogFragment`, we might just override `onCreateView()` to provide the contents of the dialog. The default implementation of `onCreateDialog()` would create an empty `Dialog`, to be populated with the `View` returned by `onCreateView()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In our `PresentationFragment` subclass of `DialogFragment`, though, we need to override `onCreateDialog()` to use a `Presentation` instead of a `Dialog`... if we have a `Presentation` to work with:

```
package com.commonsware.android.preso.fragment;

import android.app.Dialog;
import android.app.DialogFragment;
import android.app.Presentation;
import android.content.Context;
import android.os.Bundle;
import android.view.Display;

abstract public class PresentationFragment extends DialogFragment {
  private Display display=null;
  private Presentation preso=null;

  @Override
  public Dialog onCreateDialog(Bundle savedInstanceState) {
    if (preso == null) {
      return(super.onCreateDialog(savedInstanceState));
    }

    return(preso);
  }

  public void setDisplay(Context ctxt, Display display) {
    if (display == null) {
      preso=null;
    }
    else {
      preso=new Presentation(ctxt, display, getTheme());
    }

    this.display=display;
  }

  public Display getDisplay() {
    return(display);
  }

  protected Context getContext() {
    if (preso != null) {
      return(preso.getContext());
    }

    return(getActivity());
  }
}
```

We also expose `getDisplay()` and `setDisplay()` accessors, to supply the `Display` object to be used if this fragment will be thrown onto an external display.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`setDisplay()` also creates the `Presentation` object wrapped around the display, using the three-parameter `Presentation` constructor that supplies the theme to be used (in this case, using the `getTheme()` method, which a subclass could override if desired).

`PresentationFragment` also implements a `getContext()` method. If this fragment will be used with a `Display` and `Presentation`, this will return the `Context` from the `Presentation`. If not, it returns the `Activity` associated with this `Fragment`.

This project contains a `WebPresentationFragment`, that pours the same basic Android source code used elsewhere in this book for a `WebViewFragment` into a subclass of `PresentationFragment`:

```
package com.commonsware.android.preso.fragment;

import android.annotation.TargetApi;
import android.os.Build;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.webkit.WebView;

public class WebPresentationFragment extends PresentationFragment {
  private WebView mWebView;
  private boolean mIsWebViewAvailable;

  /**
   * Called to instantiate the view. Creates and returns the
   * WebView.
   */
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    if (mWebView != null) {
      mWebView.destroy();
    }

    mWebView=new WebView(getContext());
    mIsWebViewAvailable=true;
    return mWebView;
  }

  /**
   * Called when the fragment is visible to the user and
   * actively running. Resumes the WebView.
   */
  @TargetApi(11)
  @Override
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  public void onPause() {
    super.onPause();

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
      mWebView.onPause();
    }
  }

  /**
   * Called when the fragment is no longer resumed. Pauses
   * the WebView.
   */
  @TargetApi(11)
  @Override
  public void onResume() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
      mWebView.onResume();
    }

    super.onResume();
  }

  /**
   * Called when the WebView has been detached from the
   * fragment. The WebView is no longer available after this
   * time.
   */
  @Override
  public void onDestroyView() {
    mIsWebViewAvailable=false;
    super.onDestroyView();
  }

  /**
   * Called when the fragment is no longer in use. Destroys
   * the internal state of the WebView.
   */
  @Override
  public void onDestroy() {
    if (mWebView != null) {
      mWebView.destroy();
      mWebView=null;
    }
    super.onDestroy();
  }

  /**
   * Gets the WebView.
   */
  public WebView getWebView() {
    return mIsWebViewAvailable ? mWebView : null;
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

(and, as noted in Tutorial #9, where `WebViewFragment` was introduced, the flawed comments came from the original Android open source code from which this fragment was derived)

The only significant difference, besides the superclass, is that the `onCreateView()` method uses `getContext()`, not `getActivity()`, as the `Context` to use when creating the `WebView`.

And, the project has a `SamplePresentationFragment` subclass of `WebPresentationFragment`, where we use the factory-method-and-arguments pattern to pass a URL into the fragment to use for populating the `WebView`:

```
package com.commonsware.android.preso.fragment;

import android.content.Context;
import android.os.Bundle;
import android.view.Display;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class SamplePresentationFragment extends WebPresentationFragment {
  private static final String ARG_URL="url";

  public static SamplePresentationFragment newInstance(Context ctxt,
                                                       Display display,
                                                       String url) {
    SamplePresentationFragment frag=new SamplePresentationFragment();

    frag.setDisplay(ctxt, display);

    Bundle b=new Bundle();

    b.putString(ARG_URL, url);
    frag.setArguments(b);

    return(frag);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=
        super.onCreateView(inflater, container, savedInstanceState);

    getWebView().loadUrl(getArguments().getString(ARG_URL));

    return(result);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }
}
```

Using PresentationFragment

Our activity's layout now contains not only a `TextView`, but also a `FrameLayout` into which we will slot the `PresentationFragment` if there is no external display:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal"
  tools:context=".MainActivity">

  <TextView
    android:id="@+id/prose"
    android:layout_width="0px"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_weight="1"
    android:gravity="center"
    android:text="@string/secondary"
    android:textSize="40sp"/>

  <FrameLayout
    android:id="@+id/preso"
    android:layout_width="0px"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:visibility="gone"/>

</LinearLayout>
```

Note that the `FrameLayout` is initially set to have `gone` as its `visibility`, meaning that only the `TextView` will appear. Based on the widths and weights, the `TextView` will take up the full screen when the `FrameLayout` is `gone`, or they will split the screen in half otherwise.

In the `onCreate()` implementation of our activity (`MainActivity`), we inflate that layout and grab both the `TextView` and the `FrameLayout`, putting them into data members:

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_main);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    inline=findViewById(R.id.preso);
    prose=(TextView)findViewById(R.id.prose);
  }
```

Our `onResume()` method, and our `RouteCallback`, are identical to those from the previous sample. Our `handleRoute()` method is *nearly* identical to the original, as is our `onPause()` method. The difference is that we need to distinguish whether we have lost an external display (and therefore want to move the Web page into the main UI) or if we are going away entirely (and therefore just wish to clean up the external display, if any). Hence, `clearPreso()` takes a `boolean` parameter (`switchToInline`), `true` if we want to show the fragment in the main UI, `false` otherwise. And, our `onPause()` and `handleRoute()` methods pass the appropriate value to `clearPreso()`:

```
  @Override
  protected void onPause() {
    clearPreso(false);

    if (router != null) {
      router.removeCallback(cb);
    }

    super.onPause();
  }

  private void handleRoute(RouteInfo route) {
    if (route == null) {
      clearPreso(true);
    }
    else {
      Display display=route.getPresentationDisplay();

      if (route.isEnabled() && display != null) {
        if (preso == null) {
          showPreso(route);
          Log.d(getClass().getSimpleName(), "enabled route");
        }
        else if (preso.getDisplay().getDisplayId() != display.getDisplayId()) {
          clearPreso(true);
          showPreso(route);
          Log.d(getClass().getSimpleName(), "switched route");
        }
        else {
          // no-op: should already be set
        }
      }
      else {
        clearPreso(true);
        Log.d(getClass().getSimpleName(), "disabled route");
      }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }
```

`showPreso()` is called when we want to display the `Presentation` on the external display. Hence, we need to remove the `WebPresentationFragment` from the main UI if it is there:

```
  private void showPreso(RouteInfo route) {
    if (inline.getVisibility() == View.VISIBLE) {
      inline.setVisibility(View.GONE);
      prose.setText(R.string.secondary);

      Fragment f=getFragmentManager().findFragmentById(R.id.preso);

      getFragmentManager().beginTransaction().remove(f).commit();
    }

    preso=buildPreso(route.getPresentationDisplay());
    preso.show(getFragmentManager(), "preso");
  }
```

Creating the actual `PresentationFragment` is delegated to a `buildPreso()` method, which employs the `newInstance()` method on the `SamplePresentationFragment`:

```
  private PresentationFragment buildPreso(Display display) {
    return(SamplePresentationFragment.newInstance(this, display,
                                                  "http://commonsware.com"));
  }
```

`clearPreso()` is responsible for *adding* the `PresentationFragment` to the main UI, if `switchToInline` is `true`:

```
  private void clearPreso(boolean switchToInline) {
    if (switchToInline) {
      inline.setVisibility(View.VISIBLE);
      prose.setText(R.string.primary);
      getFragmentManager().beginTransaction()
                          .add(R.id.preso, buildPreso(null)).commit();
    }

    if (preso != null) {
      preso.dismiss();
      preso=null;
    }
  }
```

With an external display, the results are visually identical to the original sample. Without an external display, though, our UI is presented side-by-side:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

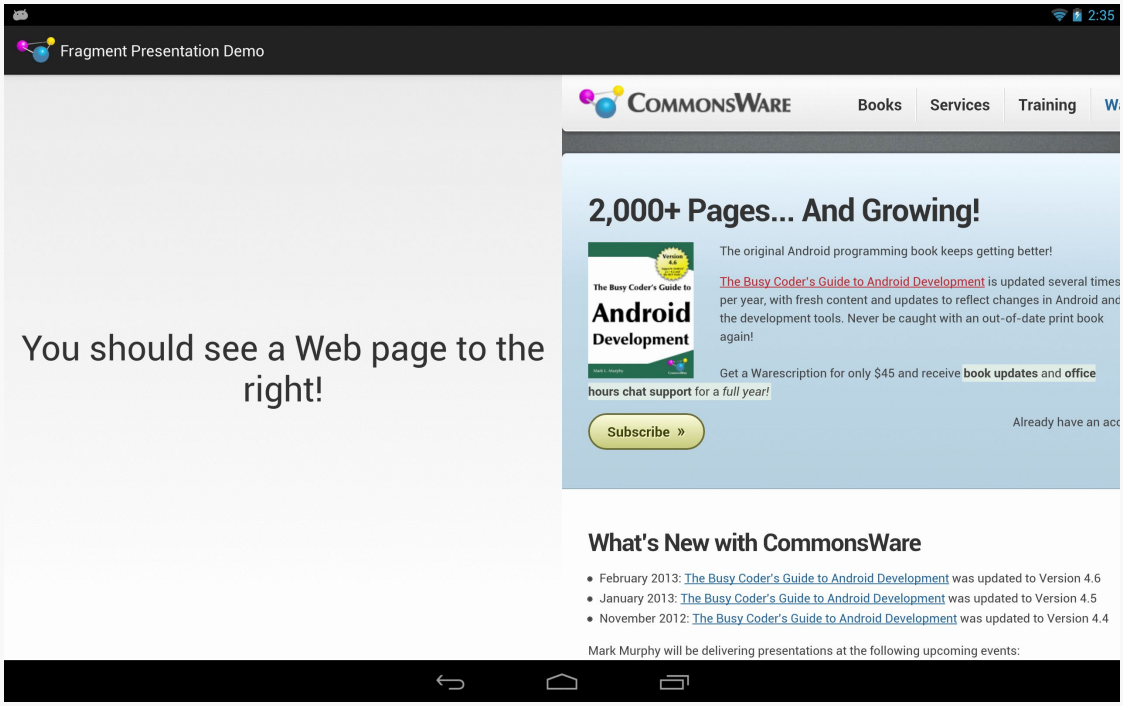


Figure 641: Nexus 10, With Inline PresentationFragment

Limits

This implementation of `PresentationFragment` has its limitations, though.

First, we cannot reuse the same fragment *instance* for both the inline UI and the `Presentation` UI, as they use different `Context` objects. Hence, production code will need to arrange to get data out of the old fragment instance and into the new instance when the screen mix changes. You might be able to leverage `onSaveInstanceState()` for that purpose, with a more-sophisticated implementation of `PresentationFragment`.

Also, depending upon the device and the external display, you *may* see multiple calls to `handleRoute()`. For example, attaching an external display may trigger three calls to your `RouteCallback`, for an attach, a detach, and another attach event. It is unclear why this occurs. However, it may require some additional logic in your app to deal with these events, if you encounter them.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Another Sample Project: Slides

At the 2013 Samsung Developer Conference, the author of this book delivered a presentation on using `Presentation`. Rather than use a traditional presentation package driven from a notebook, the author used the `Presentation/Slides` sample app. This sample app shows how to show slides on an external display, controlled by a `ViewPager` on a device's touchscreen.

What the audience saw, through most of the presentation, were simple slides. What the presenter saw was a `ViewPager`, with tabs, along with action bar items for various actions:

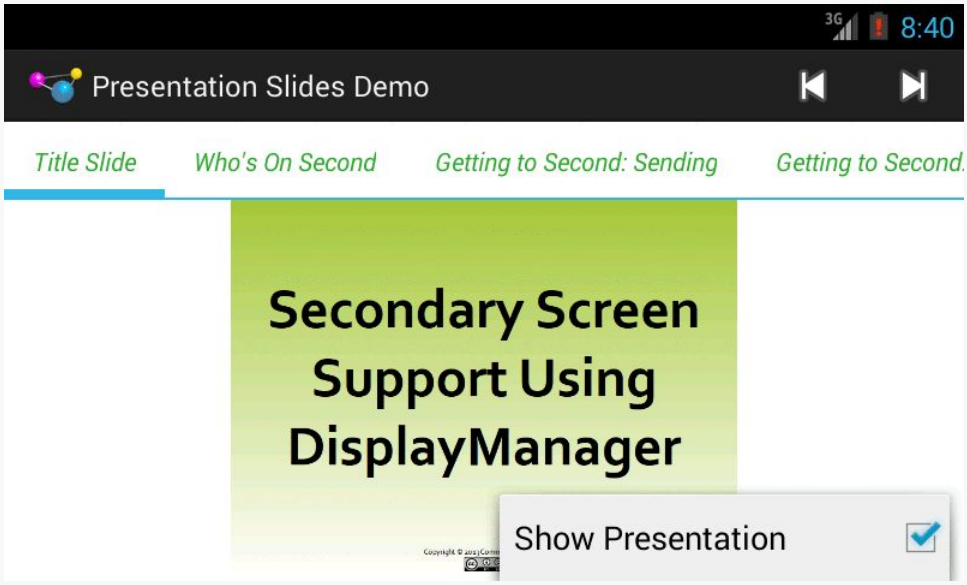


Figure 642: PresentationSlidesDemo, Showing Overflow

The Slides

The slides themselves are a series of 20 drawable resources (`img0`, `img1`, etc.), put into the `res/drawable-nodpi/` resource directory, as there is no intrinsic "density" that the slides were prepared for. As we use the slides in `ImageView` widgets, their images will be resized to fit the available `ImageView` space alone, not taking screen density into account.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

There is a matching set of 20 string resources (`title0`, `title1`, etc.) containing a string representation of the slide titles, for use with `getPageTitle()` of a `PagerAdapter`.

The PagerAdapter

That `PagerAdapter`, named `SlidesAdapter`, has each slide be visually represented by an `ImageView` widget. In this case, `SlidesAdapter` extends `PagerAdapter` directly, skipping fragments:

```
package com.commonsware.android.preso.slides;

import android.content.Context;
import android.support.v4.view.PagerAdapter;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;

class SlidesAdapter extends PagerAdapter {
  private static final int[] SLIDES= { R.drawable.img0,
      R.drawable.img1, R.drawable.img2, R.drawable.img3,
      R.drawable.img4, R.drawable.img5, R.drawable.img6,
      R.drawable.img7, R.drawable.img8, R.drawable.img9,
      R.drawable.img10, R.drawable.img11, R.drawable.img12,
      R.drawable.img13, R.drawable.img14, R.drawable.img15,
      R.drawable.img16, R.drawable.img17, R.drawable.img18,
      R.drawable.img19 };
  private static final int[] TITLES= { R.string.title0,
      R.string.title1, R.string.title2, R.string.title3,
      R.string.title4, R.string.title5, R.string.title6,
      R.string.title7, R.string.title8, R.string.title9,
      R.string.title10, R.string.title11, R.string.title12,
      R.string.title13, R.string.title14, R.string.title15,
      R.string.title16, R.string.title17, R.string.title18,
      R.string.title19 };
  private Context ctxt=null;

  SlidesAdapter(Context ctxt) {
    this.ctxt=ctxt;
  }

  @Override
  public Object instantiateItem(ViewGroup container, int position) {
    ImageView page=new ImageView(ctxt);

    page.setImageResource(getPageResource(position));
    container.addView(page,
                      new ViewGroup.LayoutParams(

ViewGroup.LayoutParams.MATCH_PARENT,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
ViewGroup.LayoutParams.MATCH_PARENT));

    return(page);
  }

  @Override
  public void destroyItem(ViewGroup container, int position,
                          Object object) {
    container.removeView((View)object);
  }

  @Override
  public int getCount() {
    return(SLIDES.length);
  }

  @Override
  public boolean isViewFromObject(View view, Object object) {
    return(view == object);
  }

  @Override
  public String getPageTitle(int position) {
    return(ctxt.getString(TITLES[position]));
  }

  int getPageResource(int position) {
    return(SLIDES[position]);
  }
}
```

The data for the `SlidesAdapter` consists of a pair of static `int` arrays, one holding the drawable resource IDs, one holding the string resource IDs.

Of note, `SlidesAdapter` has a `getPageResource()` method, to return the drawable resource ID for a given page position, which is used by `instantiateItem()` for populating the position's `ImageView`.

The PresentationFragment

We also want to be able to show the slide on an external display via a `Presentation`. As with the preceding sample app, this one uses a `PresentationFragment`, here named `SlidePresentationFragment`:

```
package com.commonsware.android.preso.slides;

import android.content.Context;
import android.os.Bundle;
import android.view.Display;
import android.view.LayoutInflater;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import com.commonsware.cwac.preso.PresentationFragment;

public class SlidePresentationFragment extends PresentationFragment {
  private static final String KEY_RESOURCE="r";
  private ImageView slide=null;

  public static SlidePresentationFragment newInstance(Context ctxt,
                                                      Display display,
                                                      int initialResource) {
    SlidePresentationFragment frag=new SlidePresentationFragment();

    frag.setDisplay(ctxt, display);

    Bundle b=new Bundle();

    b.putInt(KEY_RESOURCE, initialResource);
    frag.setArguments(b);

    return(frag);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    slide=new ImageView(getContext());

    setSlideContent(getArguments().getInt(KEY_RESOURCE));

    return(slide);
  }

  void setSlideContent(int resourceId) {
    slide.setImageResource(resourceId);
  }
}
```

Here, in addition to the sort of logic seen in the preceding sample app, we also need to teach the fragment which image it should be showing at any point in time. We do this in two ways:

1. We pass in an `int` named `initialResource` to the factory method, where `initialResource` represents the image to show when the fragment is first displayed. That value is packaged into the arguments `Bundle`, and `onCreateView()` uses that value.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. Actually putting the drawable resource into the `ImageView` for this `Presentation` is handled by `setSlideContent()`. This is called by `onCreateView()`, passing in the `initialResource` value.

The Activity

The rest of the business logic for this application can be found in its overall entry point, `MainActivity`.

Setting Up the Pager

`onCreate()` of `MainActivity` is mostly focused on setting up the `ViewPager`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  TabPageIndicator tabs=(TabPageIndicator)findViewById(R.id.titles);

  pager=(ViewPager)findViewById(R.id.pager);
  adapter=new SlidesAdapter(this);
  pager.setAdapter(adapter);
  tabs.setViewPager(pager);
  tabs.setOnPageChangeListener(this);

  helper=new PresentationHelper(this, this);
}
```

The `ViewPager` and our `SampleAdapter` are saved in data members of the activity, for later reference. We also wire in a `TabPageIndicator`, from the ViewPagerIndicator library, and arrange to get control in our `OnPageChangeListener` methods when the page changes (whether via the tabs or via a swipe on the `ViewPager` itself).

`onCreate()` also hooks up a `PresentationHelper`, following the recipe used elsewhere in this chapter. And, as `PresentationHelper` requires, we forward along the `onResume()` and `onPause()` events to it:

```
@Override
public void onResume() {
  super.onResume();
  helper.onResume();
}

@Override
public void onPause() {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    helper.onPause();
    super.onPause();
  }
```

Setting Up the Presentation

In the `showPreso()` method, required by the `PresentationHelper.Listener` interface, we create an instance of `SlidePresentationFragment`, passing in the resource ID of the current slide, as determined by the `ViewPager`:

```
  @Override
  public void showPreso(Display display) {
    int drawable=adapter.getPageResource(pager.getCurrentItem());

    preso=
        SlidePresentationFragment.newInstance(this, display, drawable);
    preso.show(getFragmentManager(), "preso");
  }
```

We then `show()` the `PresentationFragment`, causing it to appear on the attached `Display`.

The corresponding `clearPreso()` method follows the typical recipe of calling `dismiss()` on the `PresentationFragment`, if one exists:

```
  @Override
  public void clearPreso(boolean showInline) {
    if (preso != null) {
      preso.dismiss();
      preso=null;
    }
  }
```

Controlling the Presentation

However, the `SlidesPresentationFragment` now is showing the slide that was current when the `Display` was discovered or attached. What happens if the user changes the slide, using the `ViewPager`?

In that case, our `OnPageChangeListener` `onPageSelected()` method will be called, and we can update the `SlidesPresentationFragment` to show the new slide:

```
  @Override
  public void onPageSelected(int position) {
    if (preso != null) {
      preso.setSlideContent(adapter.getPageResource(position));
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }
```

Offering an Action Bar

The activity also sets up the action bar with three items:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

  <item
    android:id="@+id/first"
    android:icon="@android:drawable/ic_media_previous"
    android:showAsAction="always"
    android:title="@string/first">
  </item>
  <item
    android:id="@+id/last"
    android:icon="@android:drawable/ic_media_next"
    android:showAsAction="always"
    android:title="@string/last">
  </item>
    <item
    android:id="@+id/present"
    android:checkable="true"
    android:checked="true"
    android:showAsAction="never"
    android:title="@string/show_presentation">
  </item>

</menu>
```

Two, `first` and `last`, simply set the `ViewPager` position to be the first or last slide, respectively. This will also update the `SlidesPresentationFragment`, as `onPageSelected()` is called when we call `setCurrentItem()` on the `ViewPager`.

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.activity_actions, menu);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case R.id.present:
        boolean original=item.isChecked();

        item.setChecked(!original);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        if (original) {
          helper.disable();
        }
        else {
          helper.enable();
        }

        break;

      case R.id.first:
        pager.setCurrentItem(0);
        break;

      case R.id.last:
        pager.setCurrentItem(adapter.getCount() - 1);
        break;
    }

    return(super.onOptionsItemSelected(item));
  }
```

The other action bar item, `present`, is a checkable action bar item, initially set to be checked. This item controls what we are showing on the external display:

- If it is checked, we want to show our `Presentation`
- If it is unchecked, we want to revert to default mirroring

The theory here is that, in a presentation, we could switch from showing the slides to showing the audience what the presenter has been seeing all along.

Switching between `Presentation` and default mirroring is a matter of calling `enable()` (to show a `Presentation`) or `disable()` (to revert to mirroring) on the `PresentationHelper`.

Device Support for Presentation

Alas, there is a problem: not all Android 4.2 devices support `Presentation`, even though they support displaying content on external displays. Non-`Presentation` devices simply support classic mirroring.

Generally speaking, it appears that devices that *shipped* with Android 4.2 and higher will support `Presentation`, assuming that they have some sort of external display support (e.g., MHL). Devices that were *upgraded* to Android 4.2 are less likely to support `Presentation`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Unfortunately, at the present time, there is no known way to detect whether or not `Presentation` will work, let alone any means of filtering on this capability in the Play Store via `<uses-feature>`. With luck, this issue will be addressed in the future.

Presentations from a Service

Since `Presentation` inherits from `Dialog`, it also "inherits" one of the limitations of `Dialog`: you can only show one from an `Activity`. In many cases, that is not a big problem. If you are using the external display as an adjunct to your own app's use of the primary touchscreen, you would be using an activity anyway. However, it does prevent one from using `Presentation` to, say, implement a video player app that plays on an external display but does not tie up the touchscreen, so the user can use other apps while the video plays.

However, as it turns out, it *is* possible to drive the content of an external display from a background app... just not by using a `Presentation`.

The details of this are a bit tricky, derived from one Stack Overflow answer and another Stack Overflow question.

However, you do not need to deal with all of the details, courtesy of `PresentationService`.

`PresentationService` is a class in the CWAC-Presentation library. `PresentationService` clones some of the logic from `Presentation` and `Dialog`, enough to allow you to define a `View` that will be shown on external display, driven by a `Service`. `PresentationService` is an `abstract` base class for you to extend, where `PresentationService` handles showing your content on an external display, and you simply manage that content.

The CWAC-Presentation library has a `demoService/` directory containing a sample use of `PresentationService`. The recipe is fairly simple and is outlined in the following sections.

Step #1: Attach the Libraries

The CWAC-Presentation `README` contains instructions for attaching the libraries to your project, whether via Gradle dependencies, downloading a pair of JARs, or using the source form of the Android library project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Step #2: Add an Icky Permission

Perhaps the biggest limitation at this time for `PresentationService` is that you need to request the `SYSTEM_ALERT_WINDOW` permission. This permission is what allows services to draw windows over top of the foreground UI. In this case, we actually are not doing that — we are drawing on an external display. However, the check for this permission does not take this into account, so we need it, at least until and unless some future version of Android removes this requirement.

Step #3: Create a Stub PresentationService

As is noted above, `PresentationService` is an `abstract` class, so you will need to create your own concrete subclass of it, under whatever name you wish. And, as with any service, you will need a `<service>` element in the manifest. None of this is especially unusual.

The sample app is a service-based rendition of the `Presentation/Slides` sample app described earlier in this chapter. It has a `SlideshowService` that will display the slideshow on an external display from the background, switching slides every five seconds.

Step #4: Return the Theme

One of the `abstract` methods that you will need to implement is `getThemeId()`. This should return the value of the style resource that represents the theme that you wish to use for the widgets you are going to show on the external display.

For example, if your project uses the `@style/AppTheme` approach that is code-generated for you, you can simply return `R.style.AppTheme` from `getThemeId()`, as the sample app does:

```
  @Override
  protected int getThemeId() {
    return(R.style.AppTheme);
  }
```

Step #5: Build the View

The other `abstract` method you need to implement is `buildPresoView()`. You are passed a `Context` and a `LayoutInflater`, and your job is to use those to build your

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

UI for the external display, returning the root view. The `LayoutInflater` is already set up to use the theme you provided via `getThemeId()`.

Since this will be called shortly before showing the result on the external display, you can also take this time to initialize other aspects of your presentation. For example, the `SlideshowService` implements `Runnable` and has a `Handler` for the main application thread, initialized in `onCreate()`:

```
  private Handler handler=null;

  @Override
  public void onCreate() {
    handler=new Handler(Looper.getMainLooper());
    super.onCreate();
  }
```

`buildPresoView()` not only returns an `ImageView` for the slides, but also calls `run()`, which populates the `ImageView` and calls `postDelayed()` on the `Handler` to schedule `run()` to be called again in five seconds, thereby arranging to update the slide every five seconds:

```
  @Override
  protected View buildPresoView(Context ctxt, LayoutInflater inflater) {
    iv=new ImageView(ctxt);
    run();

    return(iv);
  }

  @Override
  public void run() {
    iv.setImageResource(SLIDES[iteration % SLIDES.length]);
    iteration+=1;

    handler.postDelayed(this, 5000);
  }
```

`onDestroy()` calls `removeCallbacks()` to break the `Handler` `postDelayed()` loop:

```
  @Override
  public void onDestroy() {
    handler.removeCallbacks(this);

    super.onDestroy();
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Step #6: Start and Stop the Service

Calling `startService()` on your service will then trigger the presentation. Or, more accurately, it will trigger `PresentationService` to work with a `PresentationHelper` to determine when a presentation should be shown. `PresentationService` will then use `buildPresoView()` to populate the external display. Conversely, calling `stopService()` will stop the presentation.

It is up to you to determine what is the trigger for these calls. The sample app simply starts the service immediately when run and stops the service in response to an action bar item click.

While the service is running, you are welcome to use an event bus or other means to control the contents of the presentation, by manipulating the widgets you created in `buildPresoView()`.

Note that it is safe to call `startService()` on the service multiple times, if you do not know whether the service is already running and need to ensure that it is running now.

Hey, What About Chromecast?

In February 2014, Google released a long-awaited SDK to allow anyone to write an app that connects to Chromecast, Google's streaming-media HDMI stick. A natural question coming out of that is whether `Presentation` and `DisplayManager` work with Chromecast.

The answer is: not presently.

While Chromecast may physically resemble a wireless display adapter, in truth it is its own device, running a customized mashup of Android and ChromeOS. Chromecast's strength is in playing streaming media from any source, primarily directly off of the Internet. The expectation with the Chromecast SDK is that apps are telling the Chromecast what to stream from, not streaming to the Chromecast itself. As such, the API for Chromecast is distinctly different from that of `Presentation`, and while the two both deal with what the Android device would consider an external display, they are not equivalent solutions.

More coverage of Chromecast can be found in the next chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Google Cast and Chromecast

A popular target for `MediaRouter`, in some countries, is Chromecast, Google's lightweight streaming media player for televisions and other HDMI displays. Originally, Chromecast was a "closed box", with no official support for third-party apps (and active work to block unofficial support). In early 2014, though, Google finally opened up Chromecast to developers.

This chapter covers what it takes to enable an Android app to "cast" content to a Chromecast, possibly as part of a broader external display strategy.

Prerequisites

In addition to the core chapters, you should read the chapter on `MediaRouter` before reading this chapter.

Here a Cast, There a Cast

You will see two terms used in this chapter and in the online literature regarding all of this: Chromecast and "Google Cast". Despite the similarities in their names, these are fairly distinct items.

What is Chromecast?

Chromecast, as noted earlier in this chapter, is a streaming media receiver, sold by Google under their own brand.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 643: Google Chromecast

It plugs into an HDMI port of a television or similar display, plus uses micro USB for supplying power.

However, rather than other streaming media receivers, that use Bluetooth or IR (infrared) peripherals for controlling the playback, Chromecast appears to use WiFi, designed to be controlled by a smartphone, tablet, or Chrome Web browser.

Chromecast itself runs its own OS, apparently a hybrid of Android and ChromeOS.

What is Google Cast?

Google Cast can be thought of as a control protocol for Google Cast-enabled receivers. Through a Google-supplied SDK (or other means), Google Cast client apps ("senders") can direct a Google Cast-enabled receiver to play, pause, rewind, fast-forward, etc. a stream.

Google Cast could, in theory, be "baked into" displays (such as a television), in addition to being supported by dedicated media receivers like the Chromecast.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Google Cast does assume that, in general, the media receiver runs its own OS and is capable of playing streaming media without ongoing assistance from the Google Cast client. Hence, the client is not "locked into" having to keep feeding content to the Google Cast client, allowing the user to go off and do other things with that client while playback is going on.

Common Chromecast Development Notes

Chromecast goes to sleep if it detects that it is plugged into a television or monitor that is turned off (or perhaps even not accepting input from the HDMI port the Chromecast is using). While it is in this sleep mode, it may not appear as an available route. You may need to keep the display active to allow Chromecast to work properly. A 720p-capable pico projector, such as the Vivitek Qumi series, can be a handy way to have a test display for Chromecast (or for live video media routes) at your development station, without the bulk of another monitor, if you have a handy surface to project upon.

Also, note that a Chromecast "uses Google's DNS regardless of what you have defined locally", according to a Google engineer. That will preclude you from using any local domains on an organization's own DNS server, without some tricky firewall configuration to route Google DNS requests to the in-house DNS server. Similarly, you cannot use machine names as pseudo-domain names, the way you might be able to using a regular Web browser.

Your API Choices

Chromecast offers up remote playback media routes and works with `RemotePlaybackClient`, as is discussed in the chapter on `MediaRouter`. The sample app for `RemotePlaybackClient` was tested on a Chromecast.

If you want greater control than is offered via `RemotePlaybackClient`, though, you can use the Cast SDK. This SDK is part of the Play Services framework, not part of Android itself. It also works solely with Google Cast devices, of which Chromecast is the only known example, whereas other sorts of devices are able to publish remote playback media routes. Hence, using the Cast SDK will tie you to Google Cast — and some of its restrictions, both technical and legal — but will give you greater developer control over the behavior of both the Google Cast device and your app.

This chapter will focus on the Cast SDK. See the chapter on `MediaRouter` for coverage of `RemotePlaybackClient`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Senders and Receivers

There are three major components to the Google Cast environment:

- The sources of streaming media, usually out on the Internet
- The software on the playback device that plays that streaming media ("receiver")
- The software on the control device (phone, tablet, Chrome Web browser) that directs the receiver about what to play and when ("sender")

The Sender App

The sender app is responsible for allowing the user to choose some media to play, then to control the actual playback (pause, start, stop, rewind, fast-forward, etc.).

The details of how to choose some media will depend heavily on the nature of the sender app. For example, a subscription-based streaming video service, such as Netflix, would allow the user to browse and search eligible content hosted by Netflix itself. Netflix presumably has its own Web service APIs that its own sender app would use for this purpose, and it is up to Netflix to offer a sensible UI for choosing a piece of media to watch.

Passing a reference (e.g., URL) to the receiver, and issuing control commands, will either be handled by `RemotePlaybackClient` (on Android) or via a Google-supplied SDK (for Android, iOS, or Chrome Web apps).

The Receiver

The details of how a receiver is implemented is up to the manufacturer of the Google Cast-enabled device. In the case of Chromecast, it is a version of the Chrome Web browser. In principle, the implementation could be anything; in practice, it is likely that the same basic software stack will be used, courtesy of licensing Google Cast technology from Google for streaming media devices.

Official Google Cast receiver software comes in three flavors: default, styled, and custom.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Default Receiver

The default receiver is what you get by default, as you might have guessed. If you do nothing else, your sender will be communicating with the default receiver. In effect, the default receiver is a specific Chrome Web app, running on the Chrome browser inside of the Chromecast, that is responsible for playback of your chosen media.

Other than providing the URLs to the media, plus requests to pause, start, stop, etc. the playback, you have no control over the default receiver, particularly from a look-and-feel standpoint.

Styled Receiver

A styled receiver is one where you, the developer, supply light branding information that is applied to what otherwise is the default receiver, such as a logo.

Whereas using the default receiver requires no explicit registration with Google, using the styled receiver does require you to register your sender app with Google, at which point you will be able to provide a URL pointing to a CSS file that contains the custom styles.

Custom Receiver

If you would rather replace the default receiver functionality with your own, either to offer more functionality, or to consume media types that may require additional configuration (e.g., DRM), you can create a custom receiver. This, in effect, is a Chrome Web app, where you provide not only CSS, but the HTML and JavaScript as well. This is substantially more complicated, and it requires registration with Google (as with the styled receiver). However, you have far greater control over what appears on the television.

Supported Media Types

The list of supported media types is likely to change over time. At present, Google Cast-enabled devices are supposed to support major media types, such as:

- MP4 and VP8 for video
- MP3, AAC, and Ogg Vorbis for audio
- PNG, JPEG, GIF, BMP, and WEBP for still images (e.g., photos)
- HLS and MPEG-DASH for streaming

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Cast SDK Dependencies

Using the Cast SDK to develop for Google Cast devices has a fair number of dependencies... and not just dependencies on particular libraries.

Developer Registration

If you are going to be using the default receiver, *and* you do not need to have debugging access to the device (e.g., to examine JavaScript logs from the Web rendering engine on the Google Cast device), you are welcome to develop your apps independently.

However, if you will use a styled or custom receiver, or you wish to gain debugging access to the device, you will need to register with Google.

This process will involve you agreeing to some terms of service (see below), along with paying a $5 registration fee.

The Terms of Service

The Google Cast SDK has separate Developer Terms of Service from anything else. If you are going to use the Google Cast SDK, you will be expected to agree to these terms as part of the registration processes. You are strongly encouraged to review these terms with qualified legal counsel. Failure to comply with the terms may cause your app (or, more accurately, your styled or custom receivers) to "be de-registered", presumably meaning that it will no longer work.

These terms contain some curious clauses, worth discussing with your attorney, including a requirement to adhere to a massive design checklist, controlling the look-and-feel of your sender and receiver. This includes a specific requirement for the precise icon to be used for initiating communications with the Google Cast receiver. Those agreeing to these terms are also barred from doing things that might allow others to display content on a Google Cast receiver without using the SDK or breaking through any access controls on the Google Cast device (e.g., creating an exploit that roots it).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Device Registration and Development Setup

While registering your device is optional, it may be handy for custom receivers, so that you can debug your custom HTML and JavaScript that is being rendered by the Google Cast device.

First, you should configure your device to publish its serial number to Google when it checks for Google Cast software updates. For the Chromecast, this involves using whatever means you used to configure the Chromecast in the first place for your network (e.g., the Chromecast Android app). There should be an option for "Send this Chromecast's serial number when checking for updates" — in the Chromecast Android app, this will be in the "Share Data" section of the device's settings screen.

Once you have registered as a Cast SDK developer, the Google Cast SDK Developer Console will have an option for you to "Add New Device". You will need the Google Cast device's serial number — in the case of the Chromecast, this is etched on the underside of the device.

Note that it may take some time before your device registration will be complete, as the device will not find out about the registration until it checks for another update, and there does not appear to be a way to trigger this. Hence, you may need to wait a few hours. You will know that you have access once you can successfully connect, via a Web browser, to port 9222 on the IP address of the Google Cast device. For the Chromecast, the easiest way to get that IP address is through your Chromecast configuration tool (e.g., the Chromecast Android app). Note that the Web page may not be much (e.g., "Inspectable WebContents"), but it will not return a 404 or similar error code.

If you wish to use a styled or custom receiver, you will also need to register your application, in the same Cast SDK Console area. This will be covered in a future edition of this book.

The Official Libraries

You will need the Google Play Services SDK, which you may have used already for other portions of the Play Services framework, such as GCM, Maps V2, and so on.

You will also need the same `mediarouter` Android library project covered in the chapter on `MediaRouter`, along with its dependencies (e.g., the `support-v4` library and the `appcompat` library).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The CastCompanionLibrary… Or Not

The Play Services SDK (and its dependencies) is all that you need to write Cast SDK applications. However, Google has also published the Cast Companion Library (CCL), containing a lot of helper code to make it a bit easier for you to write apps that adhere to the design checklist

Developing Google Cast Apps

Coverage of the Cast SDK, including sample apps, will be added to this chapter in a future edition of this book.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The “Ten-Foot UI”

Increasingly, Android devices are being used to drive screens that are somewhat larger than those found on your average phone or tablet:

- Many Android phones and tablets can directly deliver content to TVs, monitors, and projectors via HDMI, MHL, SlimPort, Miracast, and similar technologies
- Android devices can control the behavior of non-Android presentation engines, like Chromecast
- Some Android devices themselves use a TV or other display as their primary screen, from big names (Google and Amazon), mid-range firms (OUYA), and firms you have never heard of (various Android “HDMI sticks” available on eBay, Alibaba, etc.)

Technically, writing for these displays is a bit different than you would do for a phone or tablet. In some cases, such as with Google Cast, writing for these displays is more substantially different.

However, in all cases, the design of the UI needs to be different, owing to different physical and usage characteristics of large screens. This chapter will focus on this so-called “ten-foot UI” and help you understand what sorts of changes will need to be considered.

Prerequisites

Understanding this chapter requires that you have read the chapter on focus management.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The sample of the "leanback" UI is a revised version of a sample app profiled in the chapter on the `MediaStore` `ContentProvider`.

What is the "Ten-Foot UI"?

The "ten-foot UI" is not referring to a UI that is 3.048 meters high or 9.87789527 × 10^–17 parsecs wide.

Rather, the distance referred to by the "ten-foot UI" indicates the approximate distance between the viewer and the screen. People usually sit farther from TVs, monitors, and projectors than they do phones or tablets when using them. Partly, that is because the screens are a lot bigger, so they do not need to sit as closely. Partly, that is because often times the screens are being "used" by more than one person (e.g., an audience watching a presentation on a projector), and everybody needs to be able to see the screen.

The expression "ten-foot UI" refers to the design constraints inherent in developing user interfaces to be used across such a distance. Even though the screen may be bigger, the *apparent* screen size (or "visual angle" may be no bigger than phones or tablets, or sometimes even less. That, plus user input differences, technical differences between TVs and other displays, and so on all go into the "ten-foot UI" design guidance that UI experts give us.

Overscan

Television standards have been with us for several decades. Television sets from the dawn of television had significantly lower and more variable quality than today's devices. The delivery of the signal at the outset had significantly lower and more variable quality than today's over-the-air HDTV or cable connections. As a result of these two characteristics, the engineers devising television standards made some decisions that, while necessary at the time, add some complexity to delivering apps to televisions, in the form of overscan.

Simply put, not all televisions show exactly the same picture. Depending on device and signal, a television may show up to 12% less of the picture, as measured horizontally and vertically. Hence, the theoretical ideal screen size (e.g., 720p = 1280 x 720 pixels) may be achieved in some cases, but you may get less (e.g., 1128 x 634 pixels) in other cases.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android TV and Fire TV ignore overscan, relying upon developers to take it into account. As a result, the reported screen resolution is *not* necessarily available to you. Instead, you need to avoid putting anything important in the outer ~10% of the screen, centering the important stuff within the available space.

So, for example, you might have a background for your game (e.g., a starfield). Make sure that there is nothing essential on the background image that the user must see that is along the outside edge. Then, if part of the background is lost due to overscan, there is no particular problem.

The bigger issue, of course, is standard foreground widgets and containers. Android developers are used to being able to have layouts that work edge-to-edge, with just a minor amount of margin so text, icons, and the like do not run right into the edge of the screen. Now, you need more than a "minor amount" of margin. Google and Amazon recommend a `27dp` margin on the top and bottom sides of your activities, and a `48dp` margin on the left and right sides of your activities.

For activity and fragment layouts that are dedicated for TV presentation, you could elect to put those margins in those layouts, or add them via a theme. However, for activity and fragment layouts that may be used both for a touchscreen device (phone or tablet) and a television, adding the margin on the touchscreen device may be unsuitable.

For that, you could use dimension resources in different resource sets. Define `overscan_horizontal` and `overscan_vertical` to be both `0dp` (or whatever) in `res/values/dimens.xml`. Define them to be `48dp` and `27dp`, respectively, in `res/values-television/dimens.xml`, where `-television` is a resource set qualifier that will be used in Android TV and other TV-based Android devices. Then, you can refer to `@dimen/overscan_horizontal` and `@dimen/overscan_vertical` in activity/fragment layouts, to take overscan into account conditionally.

Navigation

Most televisions, monitors, and projectors are not touchscreens. Users will be changing what is shown either by using some sort of remote control (e.g., Fire TV, Android TV) or by using an app that runs on a touchscreen device (e.g., direct monitor connection, Chromecast).

In the remote control scenarios, in-screen navigation becomes important. Those remote controls usually focus on some sort of D-pad or arrow keys for moving focus

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

and clicking on widgets. This forces users into a sequential-access model (e.g., click "left" three times then "enter" once) rather than the random-access model that touchscreens offer.

The chapter on focus management covers these sorts of concerns. Bear in mind that getting focus management implemented properly in your app not only helps with the "ten-foot UI", but also can help other sorts of users, such as the visually impaired or motion impaired who cannot readily use touchscreens.

Also note that text input is a significant chore when you try to do it using a remote control. Hence, even to the extent it is possible, try to limit the number of places that users *have* to type into `EditText` widgets and the like in your UI. If possible, offer a way for users to do that sort of thing via a separate app on their phone or tablet, or perhaps through a Web browser that pushes the information to the TV set-top box.

Stylistic Considerations

In addition to structural issues like overscan and focus management, there are some stylistic issues that you will need to take into account when designing your ten-foot UI.

Fonts

With phones and tablets, if the user has some difficulty reading a bit of text, they can usually fix the problem just by moving their hand a bit, to bring the screen closer.

That becomes less likely of a solution as you get into larger screens. People get annoyed if they have to get up off of their sofa to squint and try to read text on a television. In a presentation setting, people may be unable to move into a better viewing position.

To combat this:

- Err on the side of larger fonts, with a medium weight (i.e., not too light or too heavy of strokes that make up the letters)
- Aim to use simpler fonts, particularly sans-serif fonts, as those tend to be more readable at a distance

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Where practical, give the user control over font size within your application, or allow some other sort of "zoom" mechanism, to help see details that they might otherwise be unable to see
- Light text on a dark background tends to be easier to read on televisions, so consider using a theme that supports this (e.g., `Theme.Holo` as opposed to `Theme.Holo.Light`)
- Use fewer words
- Use more line spacing (e.g., via `android:lineSpacingMultiplier` on a `TextView`), so descenders from one line are clearly distinguished from the tops of characters on the next line

Padding and Margins

In addition to adding more line spacing, consider adding more padding and margins to your ten-foot UIs.

Bear in mind that screen density calculations start to go astray as the user moves further away from the screen. We are used to making those calculations based on actual pixels on phones and tablets (a.k.a., "two-foot UI"). The *apparent* size of a television may be no bigger than that of a tablet, once the user's distance from the screen is taken into account. However, screen density has no good way to take that distance into account, other than by effectively hard-coding the density (e.g., Fire TV considering everything to be `-xhdpi`). Hence, particularly for padding and margins, you may need to "finesse" your values a bit on televisions and the like.

In cases where Android is directly talking to the television (e.g., HDMI/MHL from a phone or tablet, Fire TV, Android TV, Android HDMI sticks), you can use `-notouch` qualifiers on resource sets to provide different values for dimension resources.

Colors

Usually, with the ten-foot UI, we treat televisions, monitors, and projectors equally. They do differ in one key area: color management.

Televisions, for historical reasons, tend to have different color responses than do monitors or projectors. As Google puts it:

> TV screens have higher contrast and saturation levels than computer monitors

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Even to the extent that those settings could be adjusted, if the television will be used *as a television*, the factory settings may be the proper ones. Beyond that, few television owners think about changing such things.

As a result, you need to be careful with your color choices:

- Pure white (`#FFFFFF`) can cause problems, such as "ghosting", so use a very light gray (e.g., `#F1F1F1`, `#EBEBEB`) instead. Pure black (`#000000`) is not a problem.
- Aim for more muted colors, particularly in the blue/green/violet end of the color spectrum, as opposed to bright red or orange. Warm colors tend to bleed more than cool colors.

Aspect Ratio

Bear in mind that different TVs (or other displays) may have different aspect ratios. While many will be 16:9, also consider 4:3 and 21:9 (also known as 2.35:1).

The Leanback UI

In 2014, Google added the `leanback-v17` library to the Android Support package. This contains code to help you create TV-focused user interfaces. While the intention is for this library to help you create UIs for Android TV, there is nothing strictly tied to the Android TV platform in `leanback-v17`. Your user interfaces can work just fine on other TV environments (e.g., Amazon Fire TV). And they still support touchscreen events, and so they can be used on phones and tablets as well, though perhaps not optimally.

Where to Get Leanback

Android Studio users can add a dependency on the `leanback-v17` artifact from the Android Repository:

```
dependencies {
    compile 'com.nostra13.universalimageloader:universal-image-loader:1.9.3'
    compile 'com.android.support:leanback-v17:21.0.3'
}
```

This particular bit of Gradle configuration comes from the `Leanback/VideoBrowse` sample application, which will be the focus of this "leanback" UI section. This

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

project depends not only upon `leanback-v17`, but also upon the Universal Image Loader (UIL), another image-loading library, akin to Picasso.

If you read through the chapter on `MediaStore`, this sample app will seem familiar. In the `MediaStore` chapter, we created a sample app that would present a list of videos available on the device in a `ListView`, using UIL for handling the video thumbnails. The `VideoBrowse` "leanback" sample app is the same app, adjusted to use a "leanback" UI instead of a `ListView`.

BrowseFragment

The primary UI element that we get from `leanback-v17` is `BrowseFragment`. `BrowseFragment` is a fragment designed to allow browsing of a roster of content through a two-dimensional scrolling interface. There is a list of "headers" (e.g., categories of videos), and within each header is a horizontal scrolling list of items within that header.

This sort of UI pattern is fairly commonplace in TV-centric apps, as it works well with TV-style remotes:

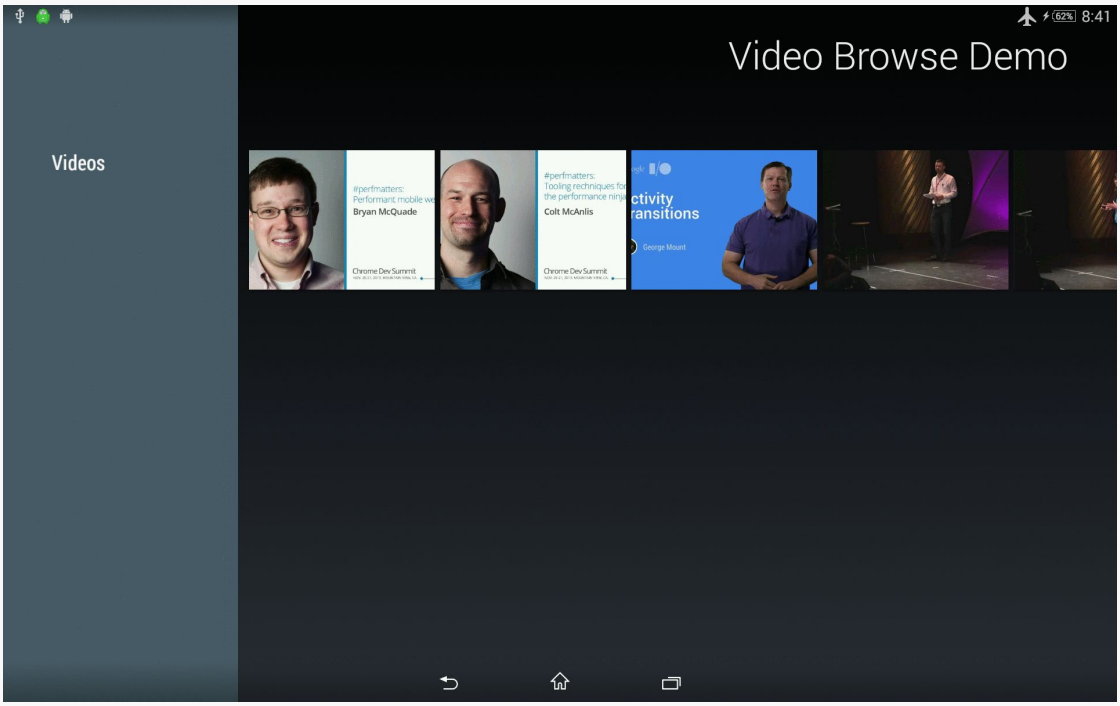


Figure 644: VideoBrowse Sample App, As Initially Launched

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The VideoBrowse sample application consists of one activity, hosting a BrowseFragment, that will display the roster of available videos on the device. Clicking on an individual video will bring up the device's default video player app, just as the VideoList sample did in the chapter on the MediaStore.

Theme and Activity

There is very little specifically required of an activity that hosts a BrowseFragment, particularly on the Java side. So long as the activity gets the BrowseFragment onto the screen, the key work is done.

In the case of our MainActivity, it uses a res/layout/main.xml file, pointing to our VideosFragment, which is a subclass of BrowseFragment:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/videos"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:name="com.commonsware.android.video.browse.VideosFragment"
/>
```

The Java code simply loads up the layout containing that static fragment, plus has an onVideoSelected() method that will be called if the user clicks on a video:

```
package com.commonsware.android.video.browse;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import java.io.File;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void onVideoSelected(String uri, String mimeType) {
    Uri video=Uri.fromFile(new File(uri));
    Intent i=new Intent(Intent.ACTION_VIEW);

    i.setDataAndType(video, mimeType);
    startActivity(i);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, there are two other requirements of the activity, in terms of what goes in the manifest:

- The activity needs to use a theme that is, or inherits from, `Theme.Leanback`
- To show up as the "launcher activity" for an Android TV device, the activity needs to have an `<action>` of `MAIN` and a `<category>` of `LEANBACK_LAUNCHER`:

```
  <application
    android:allowBackup="false"
    android:hardwareAccelerated="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="MainActivity"
      android:label="@string/app_name"

android:configChanges="keyboard|keyboardHidden|orientation|screenSize|smallestScreenSize"
      android:screenOrientation="sensorLandscape"
      android:theme="@style/Theme.Leanback">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
        <category android:name="android.intent.category.LEANBACK_LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
```

In our case, we match on either the `LAUNCHER` or the `LEANBACK_LAUNCHER` category, as this particular activity can work on either form factor family (touchscreens or TVs). However, other apps might have separate "launcher activity" implementations for phones/tablets versus televisions, and so having a a separate `LEANBACK_LAUNCHER` category allows us to indicate which activities serve which role.

This activity also sets its `screenOrientation` to `sensorLandscape`, indicating that it will always present itself in landscape mode, no matter how the device is held. It also uses a `configChanges` attribute to opt out of configuration changes due to orientation changes, as the UI is not changing in those cases.

Loading the Videos

`VideosFragment` is responsible for showing the roster of available videos on the device, using a `BrowseFragment` two-dimensional structure. This means, though, that `VideosFragment` needs to be able to find out what videos are available. As with the `VideoList` sample in the `MediaStore` chapter, `VideosFragment` will query the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

MediaStore ContentProvider to find out about the videos, by means of a CursorLoader.

In onViewCreated(), VideosFragment calls initLoader() to start loading the videos, in addition to indicating that the fragment itself will serve as the controller handling clicks on individual videos, via the setOnItemViewClickedListener() interface:

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  getLoaderManager().initLoader(0, null, this);
  setOnItemViewClickedListener(this);
}
```

For those calls to work, VideosFragment needs to implement the LoaderManager.LoaderCallbacks<Cursor> interface (for initLoader()) and the OnItemViewClickedListener (for setOnItemViewClickedListener()).

The initLoader() call triggers a call to our onCreateLoader() method, which queries the MediaStore roster of videos for all videos, ordered by title:

```
@Override
public Loader<Cursor> onCreateLoader(int arg0, Bundle arg1) {
  return(new CursorLoader(
                          getActivity(),
                          MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
                          null, null, null,
                          MediaStore.Video.Media.TITLE));
}
```

That, in turn, will eventually trigger a call to onLoadFinished():

```
@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
  mapCursorToModels(c);

  setHeadersState(BrowseFragment.HEADERS_ENABLED);
  setTitle(getString(R.string.app_name));

  ArrayObjectAdapter rows=new ArrayObjectAdapter(new ListRowPresenter());
  ArrayObjectAdapter listRowAdapter=
      new ArrayObjectAdapter(new VideoPresenter(getActivity()));

  for (Video v : videos) {
    listRowAdapter.add(v);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    HeaderItem header=new HeaderItem(0, "Videos", null);

    rows.add(new ListRow(header, listRowAdapter));
    setAdapter(rows);
  }
```

We will get to much of the code in `onLoadFinished()` a bit later in this chapter. However, the first thing that `onLoadFinished()` does is call a `mapCursorToModels()` method. This method will be responsible for taking the data from the `Cursor` we get back from `MediaStore` and using it to populate some model objects that will drive what the `BrowseFragment` displays to the user. `BrowseFragment`'s API is not especially well-suited for working with a `Cursor` directly; it is simpler to have a separate collection of model objects representing the results of the database query.

In our case, the model object will be a `Video`:

```
package com.commonsware.android.video.browse;

class Video {
  int id;
  String uri;
  String mimeType;
  String title;

  Video(int id, String uri, String mimeType, String title) {
    this.id=id;
    this.uri=uri;
    this.mimeType=mimeType;
    this.title=title;
  }

  @Override
  public String toString() {
    return(title);
  }
}
```

There are four pieces of data we need to track for the video:

- Its unique `id`, so we can get a thumbnail of the video later on
- Its `Uri` (here, held in a string representation), to be used to play back the video
- Its MIME type, also to be used to play back the video
- Its title, which will be used along with its thumbnail when rendering the video as part of the `BrowseFragment` roster of content

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

VideosFragment holds onto a collection of these Video objects in a data member named videos:

```
  private ArrayList<Video> videos=new ArrayList<Video>();
```

mapCursorToModels() iterates over the Cursor rows and creates a Video object for each row, adding the Video to the videos, and closing the Cursor when done:

```
  private void mapCursorToModels(Cursor c) {
    videos.clear();

    int idColumn=c.getColumnIndex(MediaStore.Video.Media._ID);
    int uriColumn=c.getColumnIndex(MediaStore.Video.Media.DATA);
    int mimeTypeColumn=
        c.getColumnIndex(MediaStore.Video.Media.MIME_TYPE);
    int titleColumn=
        c.getColumnIndex(MediaStore.Video.Media.TITLE);

    for (c.moveToFirst(); !c.isAfterLast(); c.moveToNext()) {
      videos.add(new Video(c.getInt(idColumn),
                           c.getString(uriColumn),
                           c.getString(mimeTypeColumn),
                           c.getString(titleColumn)));
    }

    c.close();
  }
```

Headers and Contents

So, let's look at the full onLoadFinished() method, called when we have our Cursor of videos:

```
  @Override
  public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
    mapCursorToModels(c);

    setHeadersState(BrowseFragment.HEADERS_ENABLED);
    setTitle(getString(R.string.app_name));

    ArrayObjectAdapter rows=new ArrayObjectAdapter(new ListRowPresenter());
    ArrayObjectAdapter listRowAdapter=
        new ArrayObjectAdapter(new VideoPresenter(getActivity()));

    for (Video v : videos) {
      listRowAdapter.add(v);
    }

    HeaderItem header=new HeaderItem(0, "Videos", null);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    rows.add(new ListRow(header, listRowAdapter));
    setAdapter(rows);
  }
```

As mentioned, the first thing that we do is map the `Cursor` contents to `Video` objects.

We then make two general changes to the look of the `BrowseFragment`:

- We enable the headers. In truth, that would not make sense for this particular application, as we only have one header. However, we are enabling the headers to show what that looks like, as many uses of `BrowseFragment` will need the full two-dimensional browsing experience.
- We set the title to be the `app_name` string resource. This title goes in the upper-right corner and is used to remind the user of what app they are in, much like the title in an action bar would on a phone or tablet.

We then build up the two-dimensional data model and rendering rules for the browsing experience. This involves creating instances of an `ObjectAdapter` base class, supplied by `leanback-v17`. `ObjectAdapter` fills a role reminiscent of `Adapter` with `AdapterView`, insofar as it organizes model data and helps with the rendering. However, whereas `Adapter` does that all itself, `ObjectAdapter` splits the roles out: it handles the model data and delegates to `Presenter` implementations for rendering individual items from the model data.

In the two-dimensional browsing model, we need an `ObjectAdapter` that represents our rows, where each row has a header and a nested `ObjectAdapter` for the items to appear in that row.

Just as `ArrayAdapter` is the easiest `Adapter` class to use, `ArrayObjectAdapter` is the easiest `ObjectAdapter` to use. `ArrayObjectAdapter` adapts arrays of objects, where in this case, "array" really means `ArrayList`.

Unlike `ArrayAdapter`, where we primarily build up our array and hand it to the adapter, `ArrayObjectAdapter` has us populate the "array" via methods like `add()` on the `ArrayObjectAdapter`.

So, after calling `setHeadersState()` and `setTitle()` as described above, we:

- Create an `ArrayObjectAdapter`, named `rows`, that uses the `ListRowPresenter` supplied by `leanback-v17` to render the row

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Create another `ArrayObjectAdapter`, named `listRowAdapter`, that uses a custom `VideoPresenter` that we will examine later in this chapter
- Iterate over the `Video` roster and add each to the `listRowAdapter`
- Create an instance of a `HeaderItem`, supplied by `leanback-v17`, that represents the header entry itself, with a title for that header
- Create an instance of a `ListRow`, supplied by `leanback-v17`, that wraps around the `HeaderItem` and the `listRowAdapter` for the items to show in that row
- Put the `ListRow` in the `rows` `ArrayObjectAdapter`, and pass `rows` to `setAdapter()` to tell the `BrowseFragment` what to display

A more complex app might have several `ListRow` objects in `rows`, one per header. For example, you might group videos by some sort of categorization scheme, where each `HeaderItem` names the category and the `ListRow` also contains the videos specific to that category.

Of the classes cited here, all are stock implementations from `leanback-v17`, with the exception of `VideoPresenter`, which is responsible for rendering a `Video` as an item in the horizontal list of videos.

Presenting the Presenters

A `Presenter`, in the `leanback-v17` system, is an object responsible for converting some model object (e.g., a `Video`) into a visual representation that will be used for an `ObjectAdapter`.

The `Presenter` abstract class enforces a "view holder" approach. A view holder is simply a data structure holding onto a basket of widgets. The idea is that the view holder represents all the widgets for a particular instance of the `Presenter`. So, we now have two levels of indirection over the `Adapter` approach used by `ListView` and kin: not only does `ObjectAdapter` not do the rendering, but `Presenter` alone does not do the rendering, but instead involves a view holder.

As a result, a `Presenter` implementation will tend to be artificially complex.

First, let's look at the view holder, implemented as a static class inside `VideoPresenter`, named `Holder`, that extends the stock `Presenter.ViewHolder` class:

```
  static class Holder extends Presenter.ViewHolder {
    private final ImageLoader imageLoader;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    private final ImageCardView cardView;
    private final ImageSize targetSize;

    public Holder(View view, ImageLoader imageLoader) {
      super(view);

      cardView=(ImageCardView)view;
      this.imageLoader=imageLoader;

      Resources res=view.getContext().getResources();

      targetSize=
          new ImageSize((int)res.getDimension(R.dimen.card_width),
                        (int)res.getDimension(R.dimen.card_height));
    }

    protected void updateCardViewImage(String path) {
      DisplayImageOptions opts=new DisplayImageOptions.Builder()
          .showImageOnLoading(R.drawable.ic_media_video_poster)
          .build();

      imageLoader.loadImage(path, targetSize, opts,
                            new SimpleImageLoadingListener() {
        @Override
        public void onLoadingComplete(String uri, View v, Bitmap bmp) {
          Drawable bmpDrawable=
              new BitmapDrawable(cardView.getContext().getResources(),
                                 bmp);

          cardView.setMainImage(bmpDrawable);
        }
      });
    }
  }
```

The `Presenter` will set up the UI, in the form of an `ImageCardView` – another stock class provided by `leanback-v17` that is an `ImageView` with an associated caption. As we will want to pour video thumbnails into the `ImageView`, this sample app uses the Universal Image Loader (UIL), the same way the `VideoList` sample does. Here, the `ImageLoader` instance is created by the `Presenter`, to be shared among its `Holder` instances, and we pass in the `ImageLoader` to the `Holder` and, um, hold it in a data member.

We also determine how big the thumbnail should be, creating an `ImageSize` object for UIL's use in eventually resizing the thumbnail bitmap. That `ImageSize` is based on a pair of dimension resources, `card_width` and `card_height`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="card_width">400dp</dimen>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <dimen name="card_height">300dp</dimen>
</resources>
```

The `updateCardViewImage()` will be called when we are ready to use this `ViewHolder` to present the contents of some particular `Video`. We receive the `Uri` to the video (in a `String` representation, as that is what UIL will use). We set up a UIL `DisplayImageOptions` object to indicate that we want to show a placeholder image while the thumbnail is loading.

Then, since UIL has no knowledge of how to work with an `ImageCardView` from `leanback-v17`, we have to use a different version of `loadImage()`, one that takes:

- the path to the video (for which UIL knows how to retrieve thumbnails),
- the desired thumbnail size,
- the `DisplayImageOptions`, and
- a `SimpleImageLoadingListener` that takes the `Bitmap` retrieved by UIL, wraps it in a `BitmapDrawable`, and pops that into the `ImageCardView`

Now, given the `Holder`, we can set up the rest of `VideoPresenter`, starting with its constructor:

```
  VideoPresenter(Context ctxt) {
    super();

    this.ctxt=ctxt;

    ImageLoaderConfiguration ilConfig=
        new ImageLoaderConfiguration.Builder(ctxt).build();

    imageLoader=ImageLoader.getInstance();
    imageLoader.init(ilConfig);
  }
```

Here, mostly, we are setting up the `ImageLoader` from UIL, using the same basic recipe as was used in the `VideoList` sample app.

At the point in time that the `VideoPresenter` needs to create a view holder to use for rendering an item, `onCreateViewHolder()` will be called:

```
  @Override
  public ViewHolder onCreateViewHolder(ViewGroup parent) {
    ImageCardView cardView=new ImageCardView(ctxt);

    cardView.setFocusable(true);
    cardView.setFocusableInTouchMode(true);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(new Holder(cardView, imageLoader));
  }
```

Here, we set up an `ImageCardView`, marking it as focusable both for the D-pad and for touchscreens, and wrap that in our custom `Holder`.

At the point in time when we are ready to show a `Video` using the widgets managed by the `Holder`, `onBindViewHolder()` is called:

```
  @Override
  public void onBindViewHolder(Presenter.ViewHolder viewHolder,
                               Object item) {
    Video video=(Video)item;
    Holder h=(Holder)viewHolder;
    Resources res=ctxt.getResources();

    h.cardView.setTitleText(video.toString());
    h.cardView.setMainImageDimensions((int)res.getDimension(R.dimen.card_width),

(int)res.getDimension(R.dimen.card_height));

    Uri thumbnailUri=
        ContentUris.withAppendedId(MediaStore.Video.Media.EXTERNAL_CONTENT_URI,
            video.id);

    h.updateCardViewImage(thumbnailUri.toString());
  }
```

We are passed in a generic `Object` that is the model data from our `ObjectAdapter` — in this case, it will be a `Video`, as we are using `VideoPresenter` with an `ArrayObjectAdapter` that holds the `Video` instances. We update the `ImageCardView` caption based on the title of the `Video`, then set the size of the `ImageView` based upon the same dimension resources as we used with UIL's `ImageSize`. We construct a `Uri` pointing to the video as known to `MediaStore` (given the video's id), and pass the `String` representation of that into the `Holder`, which will handle the UIL work.

There is one other abstract method that we need to override to satisfy `Presenter`: `onUnbindViewHolder()`:

```
  @Override
  public void onUnbindViewHolder(Presenter.ViewHolder viewHolder) {
    ((Holder)viewHolder).cardView.setMainImage(null);
  }
```

`onUnbindViewHolder()` can often be skipped. However, if you have memory-intensive stuff in the view holder — like the bitmap in an `ImageView` — this is a fine time to take steps to release that memory. In our case, we `null` out the image in the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

ImageCardView. Ideally, we would somehow work with UIL to arrange to recycle this Bitmap object, since all of our Bitmap objects should be the same size.

Handling Clicks

BrowseFragment automatically handles a lot of input events, such as:

- arrow key events in the list of headers, to move a selection bar up and down the list
- click events on a header, to allow navigation into the row of items for that header
- arrow key events on an item, to navigate to the next or previous item
- BACK button events on an item, to return to the list of headers

It also captures the click event on an item and routes that to the onItemClicked() method of the BrowseFragment, which we override in VideosFragment:

```
@Override
public void onItemClicked(Presenter.ViewHolder viewHolder,
                          Object o,
                          RowPresenter.ViewHolder rowViewHolder,
                          Row row) {
  Video video=(Video)o;
  ((MainActivity)getActivity()).onVideoSelected(video.uri,
                                                video.mimeType);
}
```

The Object passed as the second parameter to onItemClicked() is the Object in our ObjectAdapter for the clicked-upon item. In our case, the ObjectAdapter is our ArrayObjectAdapter wrapped around our Video objects, and so the Object passed into onItemClicked() is a Video. Given that, we can call out to the hosting activity and its onVideoSelected() to go play back the selected video.

The Results

Launching the app shows our list of headers (with just the one header), thumbnails of videos in that header, and the "Video Browse Demo" title:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

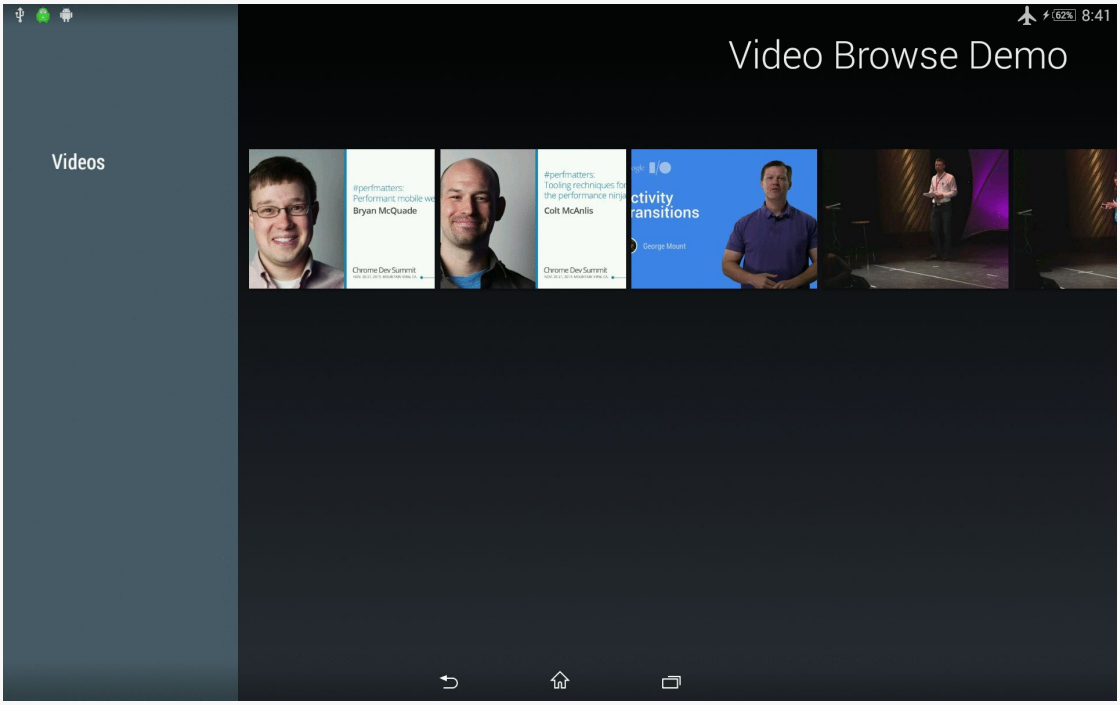


Figure 645: VideoBrowse Sample App, As Initially Launched

Selecting a video slides the headers out of the way and shows the full card for the video, including the video's title:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

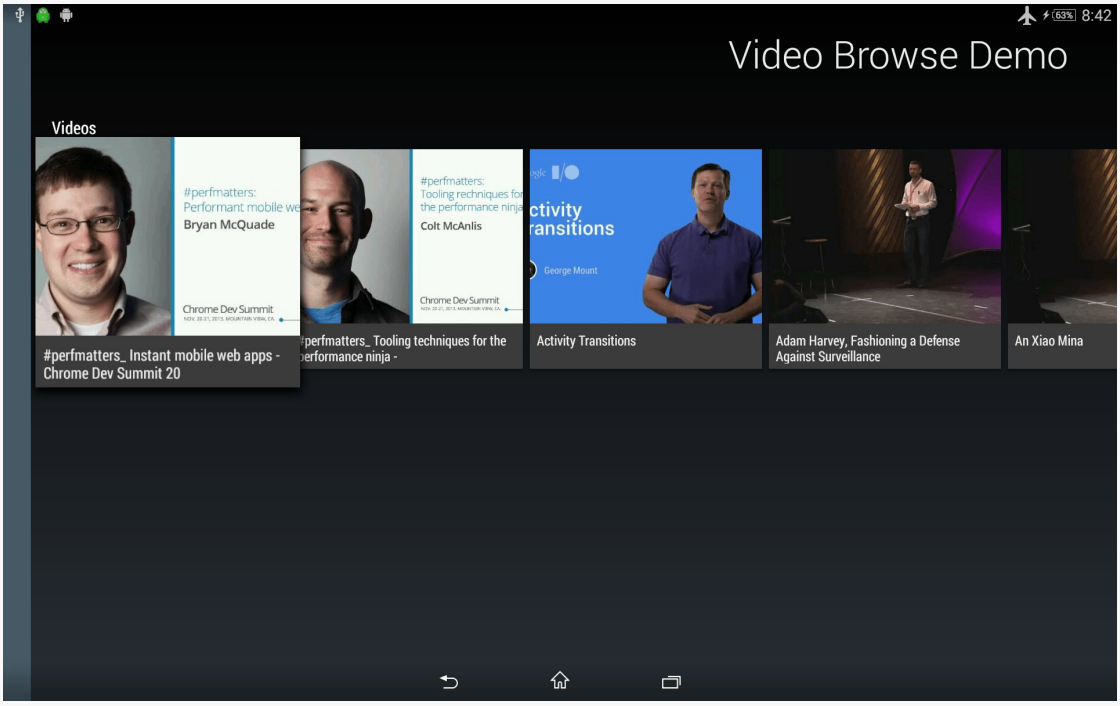


Figure 646: VideoBrowse Sample App, With First Video Selected

Clicking on the selected video brings up the default video player for the device.

Note that this UI is not tied strictly to TV-style displays. For example, the screenshots shown in this section came from an Android tablet, as you can tell by the status bar and navigation bar. The `BrowseFragment` UI is not completely out of place on a tablet, and it works with touch events as well as the key events that would be emitted by a TV-style remote control. On a phone, the `BrowseFragment` UI gets a bit cramped, particularly in portrait, though it still works.

Testing Your Theories

Ideally, you test your ten-foot UI in a ten-foot experience, using something connected to a television.

This does not have to be expensive:

- If you have a phone or tablet that can connect to a TV via HDMI, MHL, or Miracast, at most you might need a cable or a Miracast adapter

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Android "HDMI sticks" or other Android set-top boxes can be found on eBay, Alibaba, or elsewhere
- Chromecast and the Fire TV Stick are available in some markets fairly inexpensively

Any of those will cost $10–75. At the $100 price point, you can start looking at the Fire TV or Nexus Player as well.

If you need to develop using more traditional hardware (phone or a tablet) or an emulator, the big thing will be to make sure that you are estimating the screen size properly. For example, a 10" tablet, held fully at arm's length, will have the same visual angle as a modest television (~26") at a comfortable seating distance. While this will not help with color saturation, using remote controls, or other aspects of the ten-foot UI, you can at least get a sense of whether your text and UI controls will be large enough to be usable.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Putting the TVs All Together: Decktastic

This book profiles many ways of getting content to a TV:

- By means of `Presentation` and related classes, for touchscreen-enabled devices that also happen to presently have a connection to an external display
- By means of `RemotePlaybackClient`, for use with devices like the Chromecast
- By means of directly displaying output on a TV, for devices where the TV is the primary display (e.g., Android TV devices, Amazon's Fire TV and Fire TV Stick)

It is entirely possible to create one app that can support all of these modes from one code base, though you are constrained by the most limited option. In this case, `RemotePlaybackClient` is the most limited option, as its API is designed to tell some external device to play some media, whereas the other options can support comparatively arbitrary user interfaces rendered through normal Android widgets.

In this chapter, we will review the `Presentation/Decktastic` sample application. This app is designed to give the user a roster of slide-based presentations to choose from, then deliver one of those presentations. The presentation will appear on the external display (e.g., TV or projector), while the presenter will be able to control the presentation either from a touchscreen-equipped Android device or a remote control.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Prerequisites

You should read the following chapters before this one:

- Supporting External Displays
- Google Cast and ChromeCast
- The "10 Foot UI"

Reading up on specific hardware, like the Amazon Fire TV, is a good idea but not as critical.

Introducing Decktastic

Before we get into discussing the implementation of Decktastic, we should first review what the app looks like and how it functions.

Launcher UI

If you were to set up Decktastic on some test device and run it, the first thing that you would see is a media browsing UI built from the `leanback-v17` support library, showing you a roster of the available presentations to choose from:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

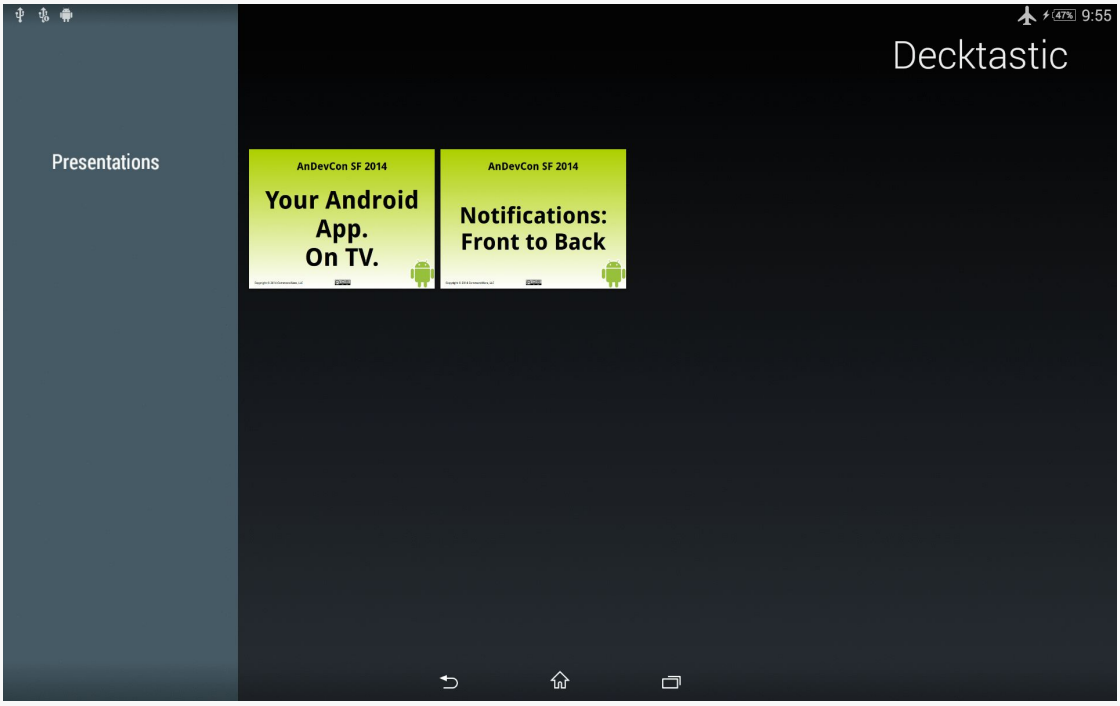


Figure 647: Decktastic Media Browser

This UI works fine on TVs and on tablets. On phones... it gets a bit cramped.

Tapping on a presentation selects it:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

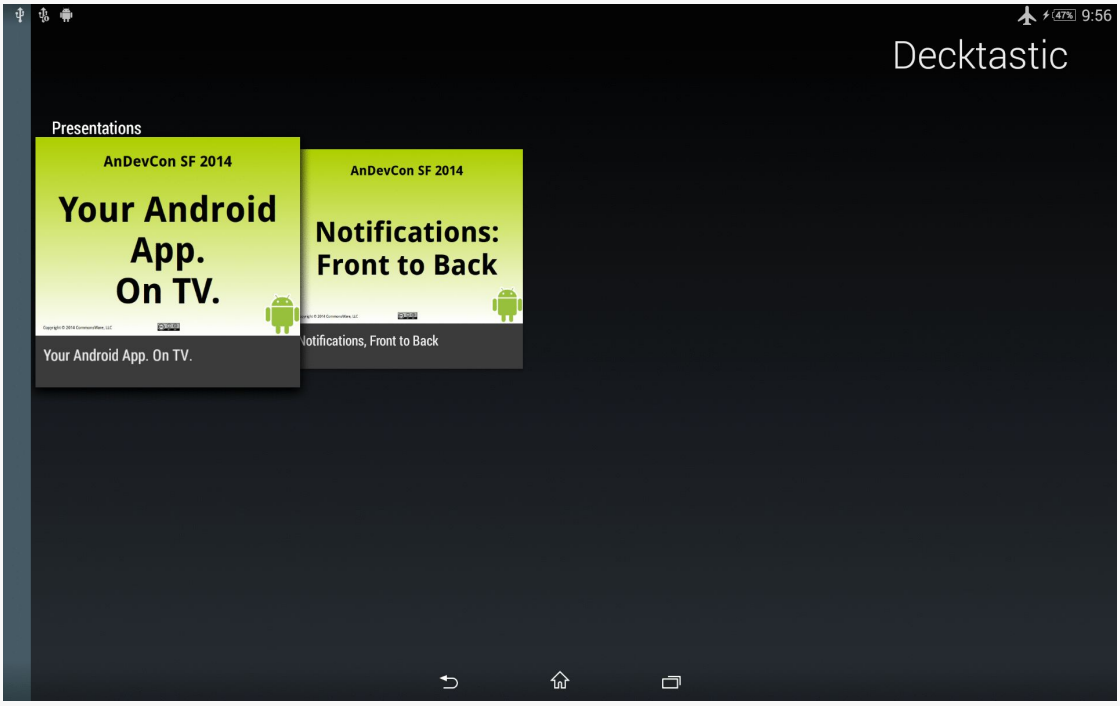


Figure 648: Decktastic Media Browser, With Selected Presentation

Presentation UI

Tapping on the presentation again opens it up into a `ViewPager`-based UI for the presenter:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 649: Decktastic Main UI, Showing Presentation and Open Overflow

However, who sees what depends a bit upon the available hardware:

- If you are running this standalone on a phone or tablet, you will see the `ViewPager`-based UI
- If you are running this on a phone or tablet with a connection to an external display, *you* will see the `ViewPager`-based UI, but the *audience* (those looking at the external display) will, by default, just see the slides
- If you are running this on a phone or tablet with a connection to a Chromecast or similar remote playback device, we get the same results as with the external display (you see the full UI, the audience sees the slides)
- If you are running this on an Android TV, Fire TV, or similar device, both you and the audience only see the slides

To move through the slides, you can:

- Swipe the `ViewPager`
- Use the `ViewPager` tabs
- Use right or down keys to move forward, or left or up keys to move backward, whether on a QWERTY keyboard (e.g., Bluetooth) or via the D-

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

pad on some form of remote (for TV-centric scenarios, like Android TV or Fire TV)

Trying Decktastic Yourself

Most of the sample applications in this book stand alone, so all you need to do is run them. Decktastic requires presentations, which means it requires a bit of setup.

If you want to see Decktastic work, you should:

- Run Decktastic once (e.g., from Android Studio). This will show that there are no available presentations. Press BACK to destroy the launcher activity.
- Download this ZIP archive containing a pair of Decktastic-compatible presentations.
- UnZIP the archive, and copy the contents (not the ZIP file itself) into `Android/data/com.commonsware.android.preso.decktastic/files` on your Android device or emulator's external storage.
- Run Decktastic again. This should give you one full presentation ("Your Android App. On TV" from the 2014 AnDevCon San Francisco conference) and one stub presentation (enough to get you a second entry in the list of presentations, but not the full slide set)

Implementing Decktastic

Decktastic is a pair of activities that "stand upon the shoulders of giants", in the form of using eight third-party libraries that provide a lot of the utility code.

The Gradle Dependencies

This project is designed to be built by Android Studio, and therefore it declares all of its dependencies in `build.gradle`. The file structure is suitable for Eclipse, but Eclipse developers would need to attach all of the appropriate dependencies manually.

The project's `build.gradle` file specifies a fair number of dependencies:

```
repositories {
    maven {
      url "http://dl.bintray.com/populov/maven"
    }
    maven {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
    mavenCentral()
}

dependencies {
    compile 'com.viewpagerindicator:library:2.4.1@aar'
    compile 'com.commonsware.cwac:presentation:0.4.+'
    compile 'com.android.support:support-v13:21.0.3'
    compile 'com.google.code.gson:gson:2.3'
    compile 'de.greenrobot:eventbus:2.4.0'
    compile 'com.commonsware.cwac:mediarouter:0.2.+'
    compile 'com.android.support:leanback-v17:21.0.3'
    compile 'com.squareup.picasso:picasso:2.4.0'
}
```

The project uses:

- Two CWAC libraries, `cwac-presentation` for `PresentationHelper` and `cwac-mediarouter` for the cross-port of the `MediaRouteActionProvider` to the native action bar
- `support-v13`, to pull in `ViewPager` and related classes
- The ViewPagerIndicator library
- Google's Gson
- greenrobot's EventBus
- Google's `leanback-v17` library, for "ten-foot UI" elements used in our launcher activity
- Square's Picasso, for asynchronously loading images

The Presentation Format

A Decktastic presentation consists of a JSON file and a series of image files. The image files are the slides themselves, perhaps exported from a traditional presentation package like LibreOffice Impress. The JSON file spells out what the image files are and their order of appearance.

For example, here is a JSON file from the stub presentation included in the aforementioned ZIP archive:

``json { "title": "Notifications, Front to Back", "duration": 70, "baseURL": "http://misc.commonsware.com/andevcon2014/pres02/", "slides": [{ "title": "(title slide)", "image": "img0.png" }, { "title": "Order of Battle", "image": "img1.png" }] } ```

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `title` is used on the initial "leanback" activity as part of displaying the available presentations.

The `duration` is how long the presentation should run, in minutes. This will be used for a countdown timer to help the presenter know how much time remains in the presentation.

The `baseURL` is a URL to a directory on a Web server somewhere that contains the same slide images as are available locally. This is needed to support `RemotePlaybackClient`, as Chromecast and similar devices need to be able to download their content over the network. We do not have an easy way to deliver that content from the phone or tablet that runs Decktastic and so we need a network-hosted copy of the slides as well. If you were willing to dispense with Chromecast support, you would not need this `baseURL` property.

The `slides` array contains JSON objects, each of which provides the title for a slide and the image associated with that slide. The title will be used for the `ViewPager` tabs, so the presenter knows the upcoming slides and can rapidly switch to a specific slide. The images, of course, are what the presenter and the audience see. Of particular importance is the first slide in the array, as this will be used as the "title slide", shown on the initial "leanback" activity.

The Model Classes

Given that JSON, we need a model class that will represent it, caching the parsed JSON so that we can use that information to render the presentation. We also need a model class that represents the collection of parsed presentations, so that we have the information necessary to render the "leanback" activity that allows the user to find the presentation to display.

There are two model classes in the book that handle this: `PresoContents` and `PresoRoster`.

PresoContents

`PresoContents` represents the parsed JSON, along with a few other bits of information about the presentation:

```
package com.commonsware.android.preso.decktastic;

import java.io.File;
import java.util.List;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public class PresoContents {
  String title;
  List<Slide> slides;
  int duration;
  String baseURL;
  File baseDir;
  int id=-1;

  static class Slide {
    String image;
    String title;
  }

  @Override
  public String toString() {
    return(title);
  }

  File getSlideImage(int position) {
    return(new File(baseDir, slides.get(position).image));
  }

  String getSlideTitle(int position) {
    return(slides.get(position).title);
  }

  String getSlideURL(int position) {
    return(baseURL+slides.get(position).image);
  }
}
```

The `title`, `slides`, `duration`, and `baseURL` fields come straight from the JSON. The `baseDir` field represents the directory in which the presentation was loaded; all images will be assumed to be relative to this directory. Finally, each presentation is given an `id`, so we can distinguish one presentation from another in our collection of presentations.

`PresoContents` also has getter methods to retrieve the local image file (`getSlideImage()`), title (`getSlideTitle()`), and remote image URL (`getSlideURL()`) for a slide given its `position` in the array of slides.

PresoRoster

`PresoRoster` is a singleton collection of the available presentations. It also contains the model logic for loading the collection of presentations and parsing the JSON to create an individual `PresoContents` object for a single presentation:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.preso.decktastic;

import android.util.Log;
import com.google.gson.Gson;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FilenameFilter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

class PresoRoster {
  private static final PresoRoster INSTANCE=new PresoRoster();
  private List<PresoContents> presos=new ArrayList<PresoContents>();

  static PresoRoster getInstance() {
    return(INSTANCE);
  }

  private PresoRoster() {}

  int getPresoCount() {
    return(presos.size());
  }

  PresoContents getPreso(int position) {
    return(presos.get(position));
  }

  PresoContents getPresoById(int id) {
    return(getPreso(id));
  }

  void load(File base) {
    base.mkdirs();

    String[] presoDirs=base.list(new FilenameFilter() {
      @Override
      public boolean accept(File orig, String name) {
        return(new File(orig, name).isDirectory());
      }
    });

    Gson gson=new Gson();

    for (String presoDir : presoDirs) {
      PresoContents c=loadPreso(gson, new File(base, presoDir));

      if (c!=null) {
        c.id=presos.size();
        presos.add(c);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      }
    }
  }

  private PresoContents loadPreso(Gson gson, File base) {
    PresoContents result=null;

    try {
      InputStream is=new FileInputStream(new File(base, "preso.json"));
      BufferedReader reader=
          new BufferedReader(new InputStreamReader(is));

      result=gson.fromJson(reader, PresoContents.class);
      result.baseDir=base;
      is.close();
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception parsing JSON", e);
    }

    return(result);
  }
}
```

The class includes:

- Getters to retrieve a presentation by index in the array of presentations (`getPreso()`) and to retrieve a presentation by the ID of the presentation (`getPresoById()`)
- A `getPresoCount()` method that indicates how many presentations were found
- A `load()` method that will scan for subdirectories under a given directory, then attempt to parse a `preso.json` file in the subdirectory (in a private `loadPreso()` method) using Gson

The result of `load()` is that the `PresoRoster` should be populated will all known presentations under the specified root directory. Note, though, that this work is done on the current thread, and therefore `load()` needs to be called on a background thread. Also note that `PresoRoster` makes no attempt at thread synchronization, and so `load()` should be called before anything attempts to use the `PresoRoster` getter methods like `getPresoCount()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Launcher Activity: LeanbackActivity

As noted previously, our launcher activity is one that implement's Google's "leanback" user interface, specifically a `BrowseFragment` for browsing media content. In this case, that content consists of the roster of available presentations.

The `LeanbackActivity` itself is fairly short:

```
package com.commonsware.android.preso.decktastic;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

public class LeanbackActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager()
          .beginTransaction()
          .add(android.R.id.content, new RosterFragment()).commit();
    }
  }

  public void showPreso(PresoContents preso) {
    startActivity(new Intent(this, MainActivity.class)
                    .putExtra(MainActivity.EXTRA_PRESO_ID,
                              preso.id));
  }
}
```

All it does is add a `RosterFragment` to the UI managed by the activity, plus add a `showPreso()` method that will be called by that `RosterFragment` when a presentation is selected. `showPreso()`, in turn, will start a separate activity (`MainActivity`), supplying `EXTRA_PRESO_ID` with the ID of the selected presentation, so `MainActivity` knows what presentation to show.

Manifest Entry

To work properly with the `leanback-v17` classes like `BrowseFragment`, `LeanbackActivity` needs to use `Theme.Leanback`, supplied by `leanback-v17`:

```
        <activity

android:name="com.commonsware.android.preso.decktastic.LeanbackActivity"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
android:configChanges="keyboard|keyboardHidden|orientation|screenSize|smallestScreenSize"
            android:label="@string/app_name"
            android:screenOrientation="sensorLandscape"
            android:theme="@style/Theme.Leanback">
          <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
            <category android:name="android.intent.category.LEANBACK_LAUNCHER"
/>
          </intent-filter>
        </activity>
```

Other noteworthy items in the `<activity>` element in the manifest include:

- Locking the screen orientation to `sensorLandscape`, as we want to stick with a landscape-style orientation, but it could either be "regular" or "reverse" landscape without issue
- Handling orientation-related configuration changes, which are not needed since we are locking the screen orientation to `sensorLandscape`, and therefore the UI does not change on an orientation change
- Having the `LEANBACK_LAUNCHER` category as an option in the `<intent-filter>`, as this will cause this activity to appear on Android TV's home screen launcher (as opposed to the `LAUNCHER` category used by normal Android devices)

RosterFragment

`RosterFragment` is a `BrowseFragment`, designed to provide the two-dimensional navigation of headers and items in a header. In this case, we will have just one header, "Presentations", containing all of the presentations found by `PresoRoster`.

In `onAttach()`, we check to see how many presentations are known about. If there are none, we make two assumptions:

1. That this is the first time we have needed to look for presentations, and
2. That there are presentations to be found

So, we fork a `LoadThread` to go load those presentations:

```
  @Override
  public void onAttach(Activity host) {
    super.onAttach(host);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (PresoRoster.getInstance().getPresoCount()==0) {
      new LoadThread(host).start();
    }
  }
```

Of course, those assumptions are a gross simplification. It could be that the user launched our `LeanbackActivity`, pressed BACK, then launched it again for some reason, and therefore the first `LoadThread` did not yet finish before we go and fork a second one. Or, it could be that there are no presentations to be found, in which case we scan unnecessarily. A production-grade version of this app should have a more sophisticated means of ensuring a one- (and only one-) time initialization.

`LoadThread` drops our thread priority to background levels, then tells the `PresoRoster` to load presentations from our app's standard spot on external storage. Then, we raise a `RosterLoadedEvent` on greenrobot's EventBus:

```
  private static class LoadThread extends Thread {
    private Context ctxt=null;

    LoadThread(Context ctxt) {
      super();

      this.ctxt=ctxt.getApplicationContext();

android.os.Process.setThreadPriority(android.os.Process.THREAD_PRIORITY_BACKGROUND);
    }

    @Override
    public void run() {
      PresoRoster.getInstance().load(ctxt.getExternalFilesDir(null));

      EventBus.getDefault().postSticky(new RosterLoadedEvent());
    }
  }
```

You will notice that we call `postSticky()`, not `post()` on the `EventBus` instance. This says that we not only want to deliver this event to any current registrants, but that the `EventBus` should cache this event and hand it to future registrants.

To respond to the `RosterLoadedEvent`, we register the `RosterFragment` on the bus in `onResume()` and unregister in `onPause()`:

```
  @Override
  public void onResume() {
    super.onResume();

    EventBus.getDefault().registerSticky(this);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void onPause() {
    EventBus.getDefault().unregister(this);

    super.onPause();
  }
```

Specifically, we register using `registerSticky()`. This, in conjunction with `postSticky()`, means that if events were sticky-posted in the past, we are delivered those immediately, in addition to future events. This will allow us to handle configuration changes — even though our activity and fragment might be destroyed on a locale change, or if our device is put into some sort of desk dock, we will get the `RosterLoadedEvent` when our fragment is created anew.

In particular, we have `onEventMainThread()` set up to listen for `RosterLoadedEvent`. At this point, we know that we have our data model and should render it on the screen:

```
  public void onEventMainThread(RosterLoadedEvent event) {
    setHeadersState(BrowseFragment.HEADERS_ENABLED);
    setTitle(getString(R.string.app_name));

    ArrayObjectAdapter rows=new ArrayObjectAdapter(new ListRowPresenter());
    PresoRoster roster=PresoRoster.getInstance();
    ArrayObjectAdapter listRowAdapter=new ArrayObjectAdapter(new
PresoPresenter());

    for (int i=0; i < roster.getPresoCount(); ++i) {
      listRowAdapter.add(roster.getPreso(i));
    }

    HeaderItem header=new HeaderItem(0, "Presentations", null);
    rows.add(new ListRow(header, listRowAdapter));

    setAdapter(rows);
  }
```

In `onEventMainThread()`, we:

- Indicate that we do want headers (though, in reality, since this app only has one header, you could easily skip the headers)
- Set the title to appear in the upper-right corner
- Create an `ArrayObjectAdapter` for the `rows` that make up the entirety of the `BrowseFragment` contents, using the standard `ListRowPresenter` for our headers and rows

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Create another `ArrayObjectAdapter`, wrapped around a `PresoPresenter`, that will manage the presentations in our one-and-only row
- Pour our `PresoContents` instances insto the `ArrayObjectAdapter` for our row
- Attach the "Presentations" title to the row via a standard `ListRow` object
- Tell the `RosterFragment` that the `rows` represents what it should render

This is all covered in greater detail in the chapter on the "ten-foot" UI.

In `onViewCreated()` of `RosterFragment`, we indicate that the `RosterFragment` itself should be the listener for click events on items (in our case, presentations):

```
@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
  super.onViewCreated(view, savedInstanceState);

  setOnItemViewClickedListener(this);
}
```

This works because `RosterFragment` implements the `OnItemViewClickedListener` interface and therefore implements the `onItemClicked()` method:

```
@Override
public void onItemClicked(Presenter.ViewHolder viewHolder,
                          Object o,
                          RowPresenter.ViewHolder rowViewHolder,
                          Row row) {
  ((LeanbackActivity)getActivity()).showPreso((PresoContents)o);
}
```

Here, we ask the hosting `LeanbackActivity` to show the clicked-upon presentation, which causes `LeanbackActivity` to launch a `MainActvity` to do just that.

PresoPresenter

The role of `PresoPresenter` is to render the individual items shown in the `BrowseFragment`. In this case, the items are `PresoContents` model objects; `PresoPresenter` will pour the presentation information into `ImageCardView` widgets. `ImageCardView` is supplied by the `leanback-v17` library and is designed to be used for rendering items in a `BrowseFragment`.

The `Presenter` abstract class — which `PresoPresenter` extends — enforces the view holder pattern. A `Presenter` is really responsible for creating and updating `Presenter.ViewHolder` instances, which in turn are responsible for updating the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

actual widgets themselves. To that end, the `PresoPresenter.Holder` static class is a subclass of `Presenter.ViewHolder`, one that is responsible for pouring a `PresoContents` into an `ImageCardView`:

```
static class Holder extends Presenter.ViewHolder {
  private ImageCardView cardView;
  private PicassoImageCardViewTarget viewTarget;

  public Holder(View view) {
    super(view);

    cardView=(ImageCardView)view;
    viewTarget=new PicassoImageCardViewTarget(cardView);
  }

  protected void updateCardViewImage(String path) {
    Picasso.with(cardView.getContext())
            .load("file://" + path)
            .resize(convertDpToPixel(cardView.getContext(), CARD_WIDTH),
                convertDpToPixel(cardView.getContext(), CARD_HEIGHT))
            .into(viewTarget);
  }
}
```

Here, we are going to use Picasso to load the initial slide off of disk and put it in the `ImageCardView`. However, Picasso has no built-in knowledge of `ImageCardView`, the way it has built-in knowledge of `ImageView`. We need to teach Picasso how to populate an `ImageCardView`. Picasso's mechanism for this is to define a `Target` implementation (`PicassoImageCardViewTarget` in this case) that is responsible for taking a loaded bitmap and updating the UI with it:

```
private static class PicassoImageCardViewTarget implements Target {
  private ImageCardView imageCardView;

  public PicassoImageCardViewTarget(ImageCardView imageCardView) {
    this.imageCardView=imageCardView;
  }

  @Override
  public void onBitmapLoaded(Bitmap bmp, Picasso.LoadedFrom lf) {
    Drawable bmpDrawable=
        new BitmapDrawable(imageCardView.getContext().getResources(),
                           bmp);

    imageCardView.setMainImage(bmpDrawable);
  }

  @Override
  public void onBitmapFailed(Drawable d) {
    imageCardView.setMainImage(d);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }

    @Override
    public void onPrepareLoad(Drawable d) {
      imageCardView.setMainImage(d);
    }
  }
```

`Target` requires implementations of:

- `onBitmapLoaded()`, where we take the image and put it as the "main image" of the `ImageCardView` by means of `setMainImage()`
- `onBitmapFailed()`, where we are given a failure `Drawable` and need to use it, once again by setting it as the `ImageCardView` main image
- `onPrepareLoad()`, where we are given a "loading" `Drawable` and need to use it, once more by setting it as the `ImageCardView` main image

The `PresoPresenter.Holder` class creates an instance of a `PicassoImageCardViewTarget` and uses that for the `into()` method of the Picasso `RequestBuilder` (created via the `with()` static method on the `Picasso` class).

The other thing interesting about our use of Picasso is in the `resize()` call. Particularly since Picasso does not know about `ImageCardView` and how big the image should be, we need to manually tell Picasso what size to make the image. Here, we hard-code the sizes of the card width and height in density-independent pixels:

```
  private static final int CARD_WIDTH=400;
  private static final int CARD_HEIGHT=300;
```

We then use a static `convertDpToPixel()` method to get the actual number of hardware pixels to use, based upon the current screen density:

```
  static int convertDpToPixel(Context ctxt, int dp) {
    float density=ctxt.getResources().getDisplayMetrics().density;

    return(Math.round((float)dp*density));
  }
```

Back up in `PresoPresenter` itself, the `Presenter` abstract class requires us to override `onCreateViewHolder()`, where we are responsible for creating a `Presenter.ViewHolder`. In the case of `PresoPresenter`, that comes in the form of the aforementioned `PresoPresenter.Holder`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@Override
public ViewHolder onCreateViewHolder(ViewGroup parent) {
  ImageCardView cardView=new ImageCardView(parent.getContext());

  cardView.setFocusable(true);
  cardView.setFocusableInTouchMode(true);

  return(new Holder(cardView));
}
```

We also have to override `onBindViewHolder()`, where we are given an eligible `Presenter.ViewHolder` and need to populate its widgets from a supplied item:

```
@Override
public void onBindViewHolder(Presenter.ViewHolder viewHolder,
                             Object item) {
  PresoContents preso=(PresoContents)item;
  Holder h=(Holder)viewHolder;

  h.cardView.setTitleText(preso.toString());
  h.cardView.setMainImageDimensions(CARD_WIDTH, CARD_HEIGHT);
  h.updateCardViewImage(preso.getSlideImage(0).getAbsolutePath());
}
```

Here, the item is a `PresoContents` and the `Presenter.ViewHolder` is a `PresoPresenter.Holder`. We update the `ImageCardView` title and image size based on the presentation, plus tell the `Holder` to update the image itself, calling the `updateCardViewImage()` method that contained our Picasso request.

Note that we are passing the density-independent pixels values (`CARD_WIDTH`, `CARD_HEIGHT`) to `setMainImageDimensions()`. Unfortunately, this method is undocumented, and so what the units of measure should be are not disclosed.

The `Presenter` abstract class also requires implementations of `onUnbindViewHolder()` (called when we should no longer be populating those widgets) and `onViewAttachedToWindow()` (called when the widgets associated with a `Presenter.ViewHolder` are now "live"):

```
@Override
public void onUnbindViewHolder(Presenter.ViewHolder viewHolder) {
  ((Holder)viewHolder).cardView.setMainImage(null);
}

@Override
public void onViewAttachedToWindow(Presenter.ViewHolder viewHolder) {
  // no-op
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Guts: MainActivity

All of the above was just to handle the launcher activity, to allow the user to choose a presentation. `MainActivity` is where we actually show the presentation itself.

This is based upon the `Presentation/Slides` sample app. profiled in the chapter on `Presentation`. Some of the basic logic, for managing the `ViewPager` and displaying the current slide on an external display, is identical to that sample. However, the Decktastic app also supports direct-to-TV devices (e.g., Fire TV, Android TV) and remote playback devices (e.g., Chromecast), and so it blends in some of the techniques covered elsewhere in this book.

Basic Setup

`onCreate()` of `MainActivity` in responsible for basic setup.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  preso=PresoRoster.getInstance()
                   .getPresoById(getIntent()
                                   .getIntExtra(EXTRA_PRESO_ID, 0));

  setContentView(R.layout.activity_main);

  pager=(ViewPager)findViewById(R.id.pager);
  helper=new PresentationHelper(this, this);

  selector=
      new MediaRouteSelector.Builder()
          .addControlCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK)
          .build();
  router=MediaRouter.getInstance(this);
  router.addCallback(selector, routeCB,
      MediaRouter.CALLBACK_FLAG_REQUEST_DISCOVERY);

  if (isDirectToTV()) {
    getActionBar().hide();
  }

  setupPager();
}
```

First, we take the `EXTRA_PRESO_ID` value received via an `Intent` extra and uses that to find the `PresoContents` object representing the presentation to be shown. That `PresoContents` object is then referenced by a data member named `preso`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Next, we load up the `activity_main` layout resource, containing our `ViewPager` and a `TabPageIndicator`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
    android:keepScreenOn="true">

  <com.viewpagerindicator.TabPageIndicator
    android:id="@+id/titles"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:visibility="gone"/>

  <android.support.v4.view.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  </android.support.v4.view.ViewPager>

</LinearLayout>
```

Then, we create a `PresentationHelper`, so that we find out when we should and should not be displaying a `Presentation`. As before, `MainActivity` itself is the `PresentationHelper.Listener` for finding out about these events. We will explore that more later in this chapter.

We then go through some logic for setting up remote playback device support (`MediaRouteSelector.Builder` and kin) and direct-to-TV device support (calling `isDirectToTV()`). Those will be explored later in this chapter, in sections on remote playback device support and direct-to-TV device support.

Finally, we call `setupPager()`, to populate our `ViewPager`.

The ViewPager

The `setupPager()` method is responsible for putting a `SlidesAdapter` into the `ViewPager` and otherwise setting things up to allow the presenter to control what slide is shown and for us to find out what slide the presenter selects:

```
  private void setupPager() {
    durationInSeconds=preso.duration * 60;

    if (rc!=null) {
      rc.setOverallDuration(durationInSeconds);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }

    adapter=new SlidesAdapter(this, preso);
    pager.setAdapter(adapter);

    TabPageIndicator tabs=(TabPageIndicator)findViewById(R.id.titles);

    tabs.setViewPager(pager);
    tabs.setOnPageChangeListener(this);

    if (isDirectToTV()) {
      tabs.setVisibility(View.GONE);
    }
    else {
      tabs.setVisibility(View.VISIBLE);
    }
  }
```

Some of this — specifically the `SlidesAdapter` and `TabPageIndicator` logic — is standard `ViewPager` setup work. The `durationInSeconds` stuff at the top is for setting up a `ReverseChronometer`, as will be discussed later in this chapter. The `isDirectToTV()` call and `if` block will be explained more in the section on direct-to-TV device support later in this chapter.

`SlidesAdapter` is a fragment-free edition of a `PagerAdapter`, as the slides are purely `ImageView` widgets:

```
package com.commonsware.android.preso.decktastic;

import android.content.Context;
import android.net.Uri;
import android.support.v4.view.PagerAdapter;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import com.squareup.picasso.Picasso;

class SlidesAdapter extends PagerAdapter {
  private PresoContents preso;
  private Context ctxt;

  SlidesAdapter(Context ctxt, PresoContents preso) {
    this.ctxt=ctxt;
    this.preso=preso;
  }

  @Override
  public Object instantiateItem(ViewGroup container, int position) {
    ImageView page=new ImageView(ctxt);

    container.addView(page,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                      new ViewGroup.LayoutParams(
ViewGroup.LayoutParams.MATCH_PARENT,
ViewGroup.LayoutParams.MATCH_PARENT));

    Picasso.with(ctxt).load(getSlideImageUri(position)).into(page);

    return(page);
  }

  @Override
  public void destroyItem(ViewGroup container, int position,
                          Object object) {
    container.removeView((View)object);
  }

  @Override
  public int getCount() {
    return(preso.slides.size());
  }

  @Override
  public boolean isViewFromObject(View view, Object object) {
    return(view == object);
  }

  @Override
  public String getPageTitle(int position) {
    return(preso.getSlideTitle(position));
  }

  Uri getSlideImageUri(int position) {
    return(Uri.fromFile(preso.getSlideImage(position)));
  }
}
```

Of note:

- `instantiateItem()` creates the `ImageView`, adds it to the supplied `container` (set to fill that container), and tells Picasso to go load the image asynchronously into the `ImageView`
- `destroyItem()` removes the `ImageView` from the container
- `getCount()` returns the number of pages, based on the number of slides in the `PresoContents` supplied to the `SlidesAdapter` via its constructor
- `getPageTitle` returns the page title, obtained from the `PresoContents` object
- `getSlideImageUri()` gets a `Uri` pointing to a local file from the `PresoContents`, for use both by `instantiateItem()` and by the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`Presentation` object that we will use for external display support (as will be covered later in this chapter)

Supporting the Direct-to-TV Scenario

To determine whether or not our activity is natively displaying on a TV-style screen, we check to see whether the device has either `FEATURE_TELEVISION` or `FEATURE_LEANBACK`, in the private `isDirectToTV()` method on `MainActivity`:

```
  private boolean isDirectToTV() {

return(getPackageManager().hasSystemFeature(PackageManager.FEATURE_TELEVISION)
        ||
getPackageManager().hasSystemFeature(PackageManager.FEATURE_LEANBACK));
  }
```

Admittedly, not all Android direct-to-TV devices may advertise that they have one of these features. In particular, minor-brand Android HDMI sticks might just be using a fairly vanilla Android device profile, culled from a tablet. There is no good way of detecting such a scenario, though Decktastic could provide some manual option (e.g., checkable action item) to go into direct-to-TV mode if this proved to be important.

We use `isDirectToTV()` in two places. First, in `onCreate()`, we hide the action bar if we are going direct to a TV:

```
    if (isDirectToTV()) {
      getActionBar().hide();
    }
```

Second, in `setupPager()`, we hide the `TabPagerIndicator` if we are going direct to a TV:

```
    if (isDirectToTV()) {
      tabs.setVisibility(View.GONE);
    }
    else {
      tabs.setVisibility(View.VISIBLE);
    }
```

This eliminates the "chrome" from our activity, leaving us with just the contents of the `ViewPager` itself, in the form of our slides. On the plus side, this gives us the visual output we want. However, it comes at a cost: there is no means for the presenter to change slides. After all, there is no touchscreen in this scenario, and so

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

even though the `ViewPager` could be swiped, that is not possible without a touchscreen.

To support standard presentation remotes and similar mechanisms, `MainActivity` overrides `onKeyDown()`:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  switch(keyCode) {
    case KeyEvent.KEYCODE_SPACE:
    case KeyEvent.KEYCODE_DPAD_RIGHT:
    case KeyEvent.KEYCODE_DPAD_DOWN:
    case KeyEvent.KEYCODE_PAGE_DOWN:
    case KeyEvent.KEYCODE_MEDIA_NEXT:
      if (pager.canScrollHorizontally(1)) {
        pager.setCurrentItem(pager.getCurrentItem()+1, true);
      }

      return(true);

    case KeyEvent.KEYCODE_DPAD_LEFT:
    case KeyEvent.KEYCODE_DPAD_UP:
    case KeyEvent.KEYCODE_PAGE_UP:
    case KeyEvent.KEYCODE_MEDIA_PREVIOUS:
      if (pager.canScrollHorizontally(-1)) {
        pager.setCurrentItem(pager.getCurrentItem()-1, true);
      }

      return(true);
  }

  return(super.onKeyDown(keyCode, event));
}
```

Here, we will advance to the next slide if the user presses:

- the space bar or the Page Down key on a QWERTY keyboard
- right or down arrow keys, D-pad buttons, or the like
- a “next” media button on a media remote

Conversely, we will return to the preceding slide if the user presses:

- the Page Up key on a QWERTY keyboard
- left or up arrow keys, D-pad buttons, or the like
- a “previous” media button on a media remote

This should allow most remotes for direct-to-TV devices to control our slides. Note that we are passing `true` as the second parameter to the `setCurrentItem()`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

method, and therefore the audience will see an animated transition to the next slide. That may or may not be desirable; an enhanced edition of Decktastic might allow that to be configured (e.g., via a checkable action item).

Note that this is still a bit limited compared to having touchscreen access, as our `onKeyDown()` method only moves a slide at a time. There is no facility to jump to an arbitrary spot, the way you could by swiping and tapping upon `ViewPager` tabs on a touchscreen.

Supporting External Displays

As noted earlier, in `onCreate()` of `MainActivity`, we create an instance of `PresentationHelper`, supplying the activity itself as both the `Context` and the `PresentationHelper.Listener` for presentation-related events:

```
    helper=new PresentationHelper(this, this);
```

That, in turn, requires us to forward along `onPause()` and `onResume()` events from our activity to the `PresentationHelper`:

```
  @Override
  public void onResume() {
    super.onResume();
    helper.onResume();
  }

  @Override
  public void onPause() {
    helper.onPause();
    super.onPause();
  }
```

We also have to implement `showPreso()` and `clearPreso()` methods to satisfy the `PresentationHelper.Listener` interface:

```
  @Override
  public void clearPreso(boolean showInline) {
    if (presoFrag != null) {
      presoFrag.dismiss();
      presoFrag=null;
    }
  }

  @Override
  public void showPreso(Display display) {
    Uri slide=adapter.getSlideImageUri(pager.getCurrentItem());
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    presoFrag=
        SlidePresentationFragment.newInstance(this, display, slide);
    presoFrag.show(getFragmentManager(), "presoFrag");
  }
```

In `showPreso()`, we obtain the `Uri` for the current slide by calling the `getSlideImageUri()` method we conveniently implemented on the `SlidesAdapter`. Then, we create an instance of a `SlidePresentationFragment`, handing it the slide `Uri`, and we `show()` that fragment. We only `dismiss()` the fragment in `clearPreso()`.

The fragment itself is a `PresentationFragment`, with an `ImageView` as the fragment's UI, populated using Picasso, with the `Uri` being transferred from the `newInstance()` factory method to the fragment itself via the arguments `Bundle`:

```
package com.commonsware.android.preso.decktastic;

import android.content.Context;
import android.net.Uri;
import android.os.Bundle;
import android.view.Display;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import com.commonsware.cwac.preso.PresentationFragment;
import com.squareup.picasso.Picasso;

public class SlidePresentationFragment extends PresentationFragment {
  private static final String KEY_URI="u";
  private ImageView slide=null;

  public static SlidePresentationFragment newInstance(Context ctxt,
                                                      Display display,
                                                      Uri slideUri) {
    SlidePresentationFragment frag=new SlidePresentationFragment();

    frag.setDisplay(ctxt, display);

    Bundle b=new Bundle();

    b.putParcelable(KEY_URI, slideUri);
    frag.setArguments(b);

    return(frag);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    slide=new ImageView(getContext());

    setSlideContent((Uri)getArguments().getParcelable(KEY_URI));

    return(slide);
  }

  void setSlideContent(Uri slideUri) {
    Picasso.with(getContext()).load(slideUri).into(slide);
  }
}
```

This arranges to show the *current* slide, for whatever the current slide is at the time `showPreso()` is called on `MainActivity`. However, we need to update this fragment to reflect changes in the current slide. To accomplish this, we set up `MainActivity` to implement the `OnPageChangeListener` interface, then call `setOnPageChangelListener()` on the `ViewPager` in `setupPager()` to have it forward page-change events to the activity. Of those events, we pay particular attention to `onPageSelected()`, updating the `SlidePresentationFragment` if there is one around:

```
  @Override
  public void onPageScrollStateChanged(int arg0) {
    // ignored
  }

  @Override
  public void onPageScrolled(int arg0, float arg1, int arg2) {
    // ignored
  }

  @Override
  public void onPageSelected(int position) {
    if (presoFrag != null) {
      presoFrag.setSlideContent(adapter.getSlideImageUri(position));
    }

    if (client!=null) {
      String url=preso.getSlideURL(position);

      client.play(Uri.parse(url), "image/png", null, 0, null, playCB);
    }
  }
```

We will get into the `client` stuff from `onPageSelected()` in the next section, as that pertains to supporting remote playback devices.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Supporting Chromecast and Remote Playback Devices

The key limitation of Chromecast and other remote playback devices is that they can only play back media that they can access. While Chromecast supports mirroring, that is handled via the `Presentation` API discussed previously; devices limited to the `RemotePlaybackClient` API need URLs to media files. That is why our JSON for the presentation contains a URL pointing to a copy of each slide up on some public Web server. To push those URLs over to the Chromecast at the appropriate points, we need to set up the `RemotePlaybackClient` system.

First, in `onCreate()`, we define a `MediaRouteSelector` for remote playback devices, set up a `MediaRouter`, and add a callback to find out about selected route changes, asking `MediaRouter` to scan for possible routes along the way:

```
    selector=
        new MediaRouteSelector.Builder()
            .addControlCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK)
            .build();
    router=MediaRouter.getInstance(this);
    router.addCallback(selector, routeCB,
        MediaRouter.CALLBACK_FLAG_REQUEST_DISCOVERY);
```

All of this is using the `mediarouter-v7` portion of the Android Support package, as the native `MediaRouter` and kin do not support remote playback devices.

Our menu resource for our action bar contains, among other things, a `MediaRouteActionProvider`:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:tools="http://schemas.android.com/tools"
      xmlns:android="http://schemas.android.com/apk/res/android"
      tools:ignore="AppCompatResource">

  <item
      android:id="@+id/countdown"
      android:actionViewClass=
        "com.commonsware.android.preso.decktastic.ReverseChronometer"
      android:showAsAction="always"
      android:title="This Should Not Be Needed">
  </item>
  <item
    android:id="@+id/route_provider"
    android:title="@string/media_route_provider"
    android:actionProviderClass=
      "com.commonsware.cwac.mediarouter.app.MediaRouteActionProvider"
    android:showAsAction="always"/>
  <item
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      android:id="@+id/first"
      android:icon="@android:drawable/ic_media_previous"
      android:showAsAction="always"
      android:title="@string/first">
  </item>
  <item
    android:id="@+id/last"
    android:icon="@android:drawable/ic_media_next"
    android:showAsAction="always"
    android:title="@string/last">
  </item>
  <item
      android:id="@+id/present"
      android:checkable="true"
      android:checked="true"
      android:showAsAction="never"
      android:title="@string/show_presentation">
  </item>

</menu>
```

As part of our work in setting up the action bar in `onCreateOptionsMenu()`, we retrieve the `MediaRouteActionProvider` and configure it with the same `MediaRouteSelector` that we used for the `MediaRouter` callback:

```
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.activity_actions, menu);

    rc=(ReverseChronometer)menu.findItem(R.id.countdown)
                               .getActionView();

    rc.setWarningDuration(5 * 60);
    rc.setOnClickListener(this);
    rc.setOnLongClickListener(this);
    rc.setTextSize(TypedValue.COMPLEX_UNIT_SP, 24);
    rc.setTextColor(Color.WHITE);

    if (durationInSeconds>0) {
      rc.setOverallDuration(durationInSeconds);
    }

    MenuItem item=menu.findItem(R.id.route_provider);
    MediaRouteActionProvider provider=
        (MediaRouteActionProvider)item.getActionProvider();

    provider.setRouteSelector(selector);

    return(super.onCreateOptionsMenu(menu));
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

(the lines in `onCreateOptionsMenu()` pertaining to the `ReverseChronometer` will be explained later in this chapter)

If the user interacts with the `MediaRouteActionProvider` and elects to connect to a remote playback device, our `MediaRouter.Callback` will be notified about the change of route:

```
private MediaRouter.Callback routeCB=new MediaRouter.Callback() {
  @Override
  public void onRouteSelected(MediaRouter router,
                              MediaRouter.RouteInfo route) {
    connect(route);
  }

  @Override
  public void onRouteUnselected(MediaRouter router,
                                MediaRouter.RouteInfo route) {
    disconnect();
  }
};
```

Here, we just delegate the `onRouteSelected()` and `onRouteUnselected()` callbacks to `connect()` and `disconnect()` methods on `MainActivity`. `MediaRouter.Callback` is an abstract class, not an interface — otherwise, we would simply have implemented the interface on `MainActivity` and bypassed this anonymous inner class instance.

The `connect()` method on `MainActivity` is responsible for sending over the current slide to the remote playback device:

```
private void connect(MediaRouter.RouteInfo route) {
  client=
      new RemotePlaybackClient(getApplicationContext(), route);

  if (client.isRemotePlaybackSupported()) {
    String url=preso.getSlideURL(pager.getCurrentItem());

    client.play(Uri.parse(url), "image/png", null, 0, null, playCB);
  }
  else {
    client=null;
  }
}
```

Here, we:

- Create an instance of `RemotePlaybackClient`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Confirm that the remote playback device supports the remote playback protocol (`isRemotePlaybackSupported()`)
- Call `play()`, passing over a URL pointing to the same slide that the `ViewPager` is showing from a local file

The `play()` call requires an `ItemActionCallback` as the last parameter. We really do not need the callback, but passing `null` does not work. So, we have a do-nothing `ItemActionCallback` named `playCB` that we use:

```
ItemActionCallback playCB=new ItemActionCallback() {
  @Override
  public void onResult(Bundle data, String sessionId,
                       MediaSessionStatus sessionStatus,
                       String itemId, MediaItemStatus itemStatus) {
  }

  @Override
  public void onError(String error, int code, Bundle data) {
  }
};
```

That will show the slide that was current as of the time the user connected to the remote playback device. We need to show a new slide when the presenter switches to a new slide, just as we did in the `Presentation` scenario. This too is handled in `onPageSelected()`, where we make the same sort of `play()` call that we did in `connect()`:

```
@Override
public void onPageSelected(int position) {
  if (presoFrag != null) {
    presoFrag.setSlideContent(adapter.getSlideImageUri(position));
  }

  if (client!=null) {
    String url=preso.getSlideURL(position);

    client.play(Uri.parse(url), "image/png", null, 0, null, playCB);
  }
}
```

Not only is `disconnect()` called from `MediaRouter.Callback`, but it is also called from `onDestroy()` of `MainActivity`, where we also remove that callback from the `MediaRouter`:

```
@Override
public void onDestroy() {
  disconnect();
  router.removeCallback(routeCB);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onDestroy();
  }
```

`disconnect()` releases the `RemotePlaybackClient` and ensures that we are back on our default route:

```
  private void disconnect() {
    if (client != null) {
      client.release();
      client=null;
    }

    router.getDefaultRoute().select();
  }
```

The net effect of all of this is that the slides will update on the remote playback device as the presenter switches slides, in addition to when the presenter connects to the remote playback device originally. We are not in control of any transition effects — we simply provide the slides, and it is up to the remote playback device to download and show them, however that device wishes.

The Rest of the Story

One common need of a presenter is to know how much time is remaining in which to deliver the presentation. Presentations are usually time-limited, to fit conference agendas and the like. The JSON structure for a presentation contains the duration of the presentation, and it would be useful to let the presenter know how much of that duration is remaining.

The chapter on custom views has a section outlining the implementation of a `ReverseChronometer` widget. `Chronometer` is a standard Android SDK class for counting up time (e.g., a stopwatch). `ReverseChronometer` is for counting *down* time.

Decktastic puts a `ReverseChronometer` in the action bar as a custom view, courtesy of our menu XML:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:tools="http://schemas.android.com/tools"
      xmlns:android="http://schemas.android.com/apk/res/android"
      tools:ignore="AppCompatResource">

  <item
      android:id="@+id/countdown"
      android:actionViewClass=
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        "com.commonsware.android.preso.decktastic.ReverseChronometer"
      android:showAsAction="always"
      android:title="This Should Not Be Needed">
  </item>
  <item
    android:id="@+id/route_provider"
    android:title="@string/media_route_provider"
    android:actionProviderClass=
      "com.commonsware.cwac.mediarouter.app.MediaRouteActionProvider"
    android:showAsAction="always"/>
  <item
      android:id="@+id/first"
      android:icon="@android:drawable/ic_media_previous"
      android:showAsAction="always"
      android:title="@string/first">
  </item>
  <item
    android:id="@+id/last"
    android:icon="@android:drawable/ic_media_next"
    android:showAsAction="always"
    android:title="@string/last">
  </item>
  <item
      android:id="@+id/present"
      android:checkable="true"
      android:checked="true"
      android:showAsAction="never"
      android:title="@string/show_presentation">
  </item>

</menu>
```

We customize the `ReverseChronometer` through a handful of lines in the `onCreateOptionsMenu()` method:

```
    rc=(ReverseChronometer)menu.findItem(R.id.countdown)
                               .getActionView();

    rc.setWarningDuration(5 * 60);
    rc.setOnClickListener(this);
    rc.setOnLongClickListener(this);
    rc.setTextSize(TypedValue.COMPLEX_UNIT_SP, 24);
    rc.setTextColor(Color.WHITE);

    if (durationInSeconds>0) {
      rc.setOverallDuration(durationInSeconds);
    }
```

Here we:

- Retrieve the `ReverseChronometer` from the action item
- Have it change to a “warning” presentation with five minutes remaining

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Set up the activity to respond to click and long-click events
- Set the appearance to be `24sp` white text

And, if we already know the presentation's overall duration, via the `durationInSeconds` data member, we pour that into the `ReverseChronometer` as well.

`durationInSeconds` is populated via a few lines at the top of `setupPager()`:

```
    durationInSeconds=preso.duration * 60;

    if (rc!=null) {
      rc.setOverallDuration(durationInSeconds);
    }
```

This way, no matter whether `setupPager()` or `onCreateOptionsMenu()` is called first, we pour the duration into the `ReverseChronometer`.

By default, that `ReverseChronometer` does nothing other than show the remaining time... which remains fixed by default. That is where the click and long-click event handlers come into play:

```
@Override
public void onClick(View v) {
  ReverseChronometer rc=(ReverseChronometer)v;

  if (rc.isRunning()) {
    rc.stop();
  }
  else {
    if (isFirstRCClick) {
      isFirstRCClick=false;
      rc.reset();
    }

    rc.run();
  }
}

@Override
public boolean onLongClick(View v) {
  ReverseChronometer rc=(ReverseChronometer)v;

  rc.reset();

  return(true);
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

There are three possibilities when the user taps on the `ReverseChronometer`:

- It was never clicked before (`isFirstRCClick` is `true`), in which case we ensure that the `ReverseChronometer` is reset to the overall duration before calling `run()` to start the countdown
- It is already running, in which case we call `stop()` to pause the countdown
- It was not already running (but was clicked before), in which case we call `run()` again to resume the countdown

This gives us what from a media standpoint would be play, pause, and resume logic.

A long-click will `reset()` the `ReverseChronometer`, returning the time remaining to the overall duration.

Our action bar also has a few other action items, handled in `onOptionsItemSelected()`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.present:
      boolean original=item.isChecked();

      item.setChecked(!original);

      if (original) {
        helper.disable();
      }
      else {
        helper.enable();
      }

      break;

    case R.id.first:
      pager.setCurrentItem(0);
      break;

    case R.id.last:
      pager.setCurrentItem(adapter.getCount() - 1);
      break;
  }

  return(super.onOptionsItemSelected(item));
}
```

Specifically:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- There is a checkable action item to determine whether or not we should be showing a `Presentation`. If this is unchecked, and an external display is attached, we still disable the `PresentationHelper`. This will cause normal display mirroring to begin, and the audience will see the same UI that the presenter does, complete with the `ViewPager`, action bar, and so on. Checking it re-enables the `PresentationHelper`, so if an external display is available, we start showing the slides again.
- The `first` and `last` action bar items are "fast-forward" and "rewind" options, allowing the presenter to quickly jump to the first or the last slide in the presentation. This happens via calls to `setCurrentItem()` on the `ViewPager`, which will in turn invoke `onPageSelected()`, causing us to update our `PresentationFragment` or remote playback device, if needed.

Note that since the direct-to-TV mode hides the action bar, none of these options are available to the presenter on a device like Android TV or a Fire TV. This will require the presenter to use something else to track the remaining time in a presentation, such as a countdown timer app running on a separate Android device.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Creating a MediaRouteProvider

As was noted earlier in the book, you can use `MediaRouter` to identify media routes, such as those published by devices like Google's Chromecast. Specifically, remote playback routes let you write apps that tell other devices, like the Chromecast, to play back media on your behalf.

However, not only can you write *clients* for remote playback routes, you can write *providers* of those routes. Perhaps you are working with a hardware manufacturer that is creating a Chromecast-like device. Perhaps you want to allow your app, running on a Fire TV or an Android HDMI stick, to be controlled by a user's phone or tablet. Or perhaps you are trying to tie Android into specialized media hardware that does not communicate by conventional means (e.g., wireless speakers that do not use normal Bluetooth profiles).

This chapter will outline how you can create code that will publish media routes to users of `MediaRouter`, so that you can then take those requests and forward them to a remote device.

Prerequisites

This chapter assumes that you have read the chapter on `MediaRouter`. Reading the chapter on the Chromecast and the Google Cast SDK would also be useful.

Terminology

For the purposes of this chapter:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- The “client device” refers to a phone or tablet that runs an app that should be able to direct what is shown on a streaming media player
- The “player device” refers to the streaming media player itself, which may or may not be running Android
- The “player app” refers to an Android app running on an Android-powered player device

DIY Chromecast

Google’s Chromecast is a nice little device. However, it has issues:

- The device itself is not especially open.
- The Cast SDK that Google encourages for writing Chromecast-enabled apps is not especially open.
- The terms and conditions for using the Cast SDK may be troublesome for many developers.
- Chromecast is not available globally.
- Chromecast is only one device, and there are plenty of other streaming media devices available that need to be considered.

Some of these issues can be mitigated by the use of `MediaRouter` and `RemotePlaybackClient` instead of the proprietary Cast SDK. You are not bound by any particular license terms (beyond the norm for Android development) and the implementation of the media framework is open.

However, to make this work, the client device needs to know how to talk to the player device.

The good news is that the media routing framework in Android supports plug-in media route providers for just this purpose. The OS ships with such a provider for the Chromecast, and you can create your own providers to talk to whatever else you would like to talk to. The user can then install a small app on their client device that implements this media route provider, and *any* apps already on their client device that use classes like `RemotePlaybackClient` will automatically be able to cast their desired content to the player device.

MediaRouteProvider

The guts of this come in the form of a `MediaRouteProvider`. Your custom subclass of `MediaRouteProvider` will:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Tell Android what general capabilities you support, such as remote playback, session management, and the like
- Advertise what sorts of content your player device is capable of playing (e.g., certain video MIME types, certain URL schemes like `http` and `rtsp`)
- Serve as the recipient of commands from `MediaRouter`, `RemotePlaybackClient`, and the like, for you to forward along asynchronously to the player device

Depending upon your use case, you could elect to keep the `MediaRouteProvider` private to your application. That way, your app can cast to the player device, but no other apps can. Or, you can make your `MediaRouteProvider` available to all apps on the device, with the media routing framework taking care of the IPC details to have those apps tell your `MediaRouteProvider` what the player device should do.

Player Device… and Maybe a Player App

Of course, this assumes the existence of some player device that is not supported by Android out of the box. Since Android only really supports Chromecast, external displays (e.g., HDMI, MHL, Miracast), and some Bluetooth options (e.g., external speakers) for media routes, there are countless player devices that need additional help. These will run the gamut from devices from major players (e.g., Amazon's Fire TV) to no-name devices (e.g., Android HDMI "sticks").

Some player devices will run Android. In that case, you would be writing a player app that would run on the player device that would be the recipient of commands sent to it from your `MediaRouteProvider` on the client device. For example, if you write a video player app, you could augment it with remote control capability driven by a `MediaRouteProvider` on a client device, turning your player app and anything it can run on (e.g., Fire TV, OUYA game console) into a Chromecast-like environment.

Some player devices will not run Android. If they offer some existing remote control over-the-air protocol, you could create a `MediaRouteProvider` that speaks that protocol. Or, perhaps the player devices are programmable, just not via Android (e.g., a Linux program for XMBC), in which case you might be able to write both ends of the communications channel.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Communications Protocol

Somehow, the data from the `MediaRouteProvider` needs to get to the player device (and, where relevant, the player app). Likely candidates include Bluetooth, regular WiFi (if both devices are on the same network), and WiFi Direct.

However, in principle, anything is possible. For example, there is nothing stopping you from sending `MediaRouteProvider` commands to some Web server out on the Internet, which forwards them to some distant location for use. That would be a bit unusual – normally, the user of the client device is controlling something she can see — but it certainly could be done.

The biggest thing to watch out for is the addressability of the media to be played back. There is little point in connecting a `MediaRouteProvider` to some player device, then not have the ability for the player device to access the media that the client device is requesting. The expected pattern is that the media is hosted in some (relatively) central location, like a Web server. However, once again, anything is possible. If you want to have some sort of server on the client device, to allow the player device to play back media from it, and you believe that you can adequate secure this, you are welcome to do so.

Creating the MediaRouteProvider

As noted earlier, the core of all of this is a custom `MediaRouteProvider`. Google supplies a sample application for creating such a `MediaRouteProvider`. However, it is overly complex, and it is undocumented.

This chapter will focus instead on the MediaRouter/RouteProvider sample project. This is a clone of the `MediaRouter/RemotePlayback` sample project covered earlier in this book, with the addition of a custom `MediaRouteProvider`.

Defining the Supported Actions

A `MediaRouteProvider` advertises — whether to its own app's `MediaRouter` or to the entire device — what sorts of actions it can perform. For example, a remote playback route provider needs to support actions like play, pause, resume, and stop of some piece of media.

The way this is handled in the media routing framework is via a series of `IntentFilter` objects.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Since IntentFilter objects do not need a Context to be created, it is safe to define them statically, if desired. That's what we do in DemoRouteProvider, a custom subclass of MediaRouteProvider. It declares a pair of static final IntentFilter objects, ifPlay and ifControl, which are then configured in a static initialization block:

```
  private static final IntentFilter ifPlay=new IntentFilter();
  private static final IntentFilter ifControl=new IntentFilter();

  static {
    ifPlay.addCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK);
    ifPlay.addAction(MediaControlIntent.ACTION_PLAY);
    ifPlay.addDataScheme("http");
    ifPlay.addDataScheme("https");
    ifPlay.addDataScheme("rtsp");

    try {
      ifPlay.addDataType("video/*");
    }
    catch (MalformedMimeTypeException e) {
      throw new RuntimeException("Exception setting MIME type", e);
    }

    ifControl.addCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK);
    ifControl.addAction(MediaControlIntent.ACTION_PAUSE);
    ifControl.addAction(MediaControlIntent.ACTION_RESUME);
    ifControl.addAction(MediaControlIntent.ACTION_STOP);
    ifControl.addAction(MediaControlIntent.ACTION_GET_STATUS);
    ifControl.addAction(MediaControlIntent.ACTION_SEEK);
  }
```

Both stipulate that the are looking for Intent objects in the MediaControlIntent.CATEGORY_REMOTE_PLAYBACK category. This category is used for all media routing Intents that form the foundation of the routing framework.

ifPlay is defined as supporting MediaControlIntent.ACTION_PLAY, stating that we know how to play back some content. The qualifications for "some content" are handled via scheme and type constraints placed on the IntentFilter. Here, we limit the content to be URLs that might be reachable by a playback device (http, https, rtsp) and have a MIME type matching video/*. Hence, we are stating that we can play back streaming video.

ifControl sets up the remaining actions that we support:

- MediaControlIntent.ACTION_PAUSE
- MediaControlIntent.ACTION_RESUME
- MediaControlIntent.ACTION_STOP

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `MediaControlIntent.ACTION_GET_STATUS`
- `MediaControlIntent.ACTION_SEEK`

These are placed on an independent `IntentFilter` because, technically, we can support these actions on any type of media. In the case of this specific example, the only media we support is streaming video. But, we could configure other `IntentFilter` objects, like `ifPlay` was, stating yet other media types that we handle.

To fully comply with the `RemotePlaybackClient` API, we must advertise that we handle all of those actions... even if our intended client will not use all of them.

We could also:

- Advertise that we support session management actions, like `MediaControlIntent.ACTION_START_SESSION`
- Advertise that we support the "enqueue" operation for stacking up media to be played (e.g., `MediaControlIntent.ACTION_ENQUEUE`) and manipulating that queue (e.g., `MediaControlIntent.ACTION_REMOVE`)
- Define a custom category for other actions that we support that are "out of band" with respect to the standard media routing actions

All of those are demonstrated in Google's sample app.

Creating the Descriptors

Just because we have some `static IntentFilter` objects does not mean that anything will pay attention to them. We need to actually register them with the media routing framework, wrapped in a pair of "descriptor" objects. `DemoRouteProvider` calls a private `handleDiscovery()` method from the constructor, where `handleDiscovery()` sets up the descriptors:

```
  private void handleDiscovery() {
    MediaRouteDescriptor.Builder mrdBuilder=
        new MediaRouteDescriptor.Builder(DEMO_ROUTE_ID, "Demo Route");

    mrdBuilder.setDescription("The description of a demo route")
              .addControlFilter(ifPlay)
              .addControlFilter(ifControl)
              .setPlaybackStream(AudioManager.STREAM_MUSIC)
              .setPlaybackType(MediaRouter.RouteInfo.PLAYBACK_TYPE_REMOTE)
              .setVolumeHandling(MediaRouter.RouteInfo.PLAYBACK_VOLUME_FIXED);

    MediaRouteProviderDescriptor.Builder mrpdBuilder=
        new MediaRouteProviderDescriptor.Builder();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    mrpdBuilder.addRoute(mrdBuilder.build());

    setDescriptor(mrpdBuilder.build());
  }
```

In the end, we need to provide a `MediaRouteProviderDescriptor` to the `MediaRouteProvider` by means of a `setDescriptor()` method. `MediaRouteProviderDescriptor` is, in effect, metadata about the `MediaRouteProvider` itself. At the present time, the only thing this holds is a set of `MediaRouteDescriptor` objects, one for each media route that the `MediaRouteProvider` claims to support.

A `MediaRouteProvider` is made up of several pieces of information, including:

- The `IntentFilter`(s) representing the supported actions and, where relevant, MIME types and schemes
- A locally-unique ID of the route, to distinguish it from any other one that we might configure
- A name and description, which the user will see when they try to connect to this route (e.g., via a `MediaRouteActionProvider`)
- What audio stream is being used for the playback, from the standpoint of volume management, audio ducking, and the like
- Whether the playback is occurring locally on the device to some peripheral (e.g., speaker) or if the playback is occurring remotely on a player device (e.g., Chromecast)
- Whether playback volume is controlled here on the client device or on the player device
- Etc.

These are all configured on a `MediaRouteProvider` by creating a `MediaRouteProvider.Builder` and supplying the values either in the `Builder` constructor or via fluent setter methods. In the particular case of our simple demo provider, we:

- Use various strings for the ID, name, and description
- Use the two `IntentFilter` objects defined earlier to indicate what actions we can perform
- Indicate that the playback stream is `STREAM_MUSIC`, that the playback type is `PLAYBACK_TYPE_REMOTE`, and that the volume handling is `PLAYBACK_VOLUME_FIXED` (i.e., volume should be managed on the TV or whatever the media is being played upon)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It is very likely that you will elect to have several `MediaRouteDescriptor` objects for different client application scenarios. Google's sample app uses a total of four `MediaRouteDescriptor` objects:

- One set up largely like the one in this sample
- One set up with `PLAYBACK_VOLUME_VARIABLE` (so volume is controllable by a client app)
- One set up with variable volume plus queuing actions
- One set up with variable volume plus queuing and session management actions

Receiving the Actions

Now, we have told the media routing framework what actions we support. Some app will then try to use `RemotePlaybackClient` and ask us to perform those actions. Hence, we need to find out when this happens, so we can do the actual work of having the playback device actually play back the media, pause the media, etc.

To do this, we need to create a custom subclass of `MediaRouteProvider.RouteController`. This contains a series of callback methods which we can override to find out when various events occur.

There are four such callback methods that the `DemoRouteController` subclass of `MediaRouteProvider.RouteController` implements:

- `onSelect()`, which will be called when a client app has selected our `MediaRouteProvider` to handle some media on behalf of that client app
- `onUnselect()` and `onRelease()`, which will be called when the client app disconnects from our `MediaRouteProvider`
- `onControlRequest()`, which will be called when some specific action that we advertised is requested, such as playing back a piece of media

The `DemoRouteController` just logs a message to LogCat for the first three callbacks:

```
@Override
public void onRelease() {
  Log.d(getClass().getSimpleName(), "released");
}

@Override
public void onSelect() {
  Log.d(getClass().getSimpleName(), "selected");
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@Override
public void onUnselect() {
  Log.d(getClass().getSimpleName(), "unselected");
}
```

The `onControlRequest()` method is a bit more complex, as *all* control requests route through here: play, pause, resume, stop, etc. `onControlRequest()` is passed the `Intent` identifying the particular action that should be performed, and we can examine the `Intent` action string to determine what needs to be done. In this case, `onControlRequest()` delegates the real work to action-specific methods like `onPlayRequest()`:

```
@Override
public boolean onControlRequest(Intent i, ControlRequestCallback cb) {
  if (i.hasCategory(MediaControlIntent.CATEGORY_REMOTE_PLAYBACK)) {
    if (MediaControlIntent.ACTION_PLAY.equals(i.getAction())) {
      return(onPlayRequest(i, cb));
    }
    else if (MediaControlIntent.ACTION_PAUSE.equals(i.getAction())) {
      return(onPauseRequest(i, cb));
    }
    else if (MediaControlIntent.ACTION_RESUME.equals(i.getAction())) {
      return(onResumeRequest(i, cb));
    }
    else if (MediaControlIntent.ACTION_STOP.equals(i.getAction())) {
      return(onStopRequest(i, cb));
    }
    else if (MediaControlIntent.ACTION_GET_STATUS.equals(i.getAction())) {
      return(onGetStatusRequest(i, cb));
    }
    else if (MediaControlIntent.ACTION_SEEK.equals(i.getAction())) {
      return(onSeekRequest(i, cb));
    }
  }

  Log.w(getClass().getSimpleName(), "unexpected control request"
      + i.toString());

  return(false);
}
```

`onControlRequest()` should return `true` if we agree to perform the action and will use the supplied `ControlRequestCallback` object to asynchronously deliver our results. If `onControlRequest()` returns `false`, that means that we are rejecting the action for some reason, such as it being one that is unrecognized. In `DemoRouteController`, that will occur if the category or the action on the `Intent` is not one of the supported options.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that if you opted into variable volume, there are `onSetVolume()` and `onUpdateVolume()` callback methods that will give you access to those events.

Handling the Actions

For those actions that you advertise and receive in `onControlRequest()`, you need to actually do the work for those actions. The details of this will vary widely depending upon your playback device and playback app that you are supporting. For example, you might establish a WiFi Direct connection in `onSelect()`, then use that connection in handling play, pause, etc. actions.

However, a few aspects of handling these actions will be in common across all implementations:

- `onControlRequest()` must return `true` or `false` as was described in the preceding section
- You must call `onResult()` or `onError()` on the `ControlRequestCallback` object to indicate if the action succeeded or failed
- You must supply an appropriate `Bundle` to those methods, particularly to `onResult()`, containing the right set of values to provide more details about the results of the action

The details of what that `Bundle` must contain are documented on the `MediaControlIntent` class, on the definition of each action string (e.g., `ACTION_PLAY`).

With that in mind, let's look at the six actions supported by `DemoRouteController`.

Play

The `Bundle` passed to `onResult()` of the `ControlRequestCallback`, when the action is `ACTION_PLAY`, needs three values:

- `EXTRA_SESSION_ID`: if you are implementing session management, this will be the unique session ID (`String`) for the session you are playing the media in. If you are not implementing session management, then what you are supposed to return is undocumented and (hopefully) unused
- `EXTRA_ITEM_ID`: if you are implementing "enqueue" support, this will be the item ID (`String`) for managing this item in the queue of available items. If you are not supporting a playback queue, then what you are supposed to return is undocumented and (hopefully) unused

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `EXTRA_ITEM_STATUS`: this should point to a `Bundle` created from a `MediaItemStatus` object where you indicate what the status is of the playback of this item

You create a `MediaItemStatus` object via a `MediaItemStatus.Builder`, where you can pass into the constructor a value indicating the overall status (e.g., `MediaItemStatus.PLAYBACK_STATE_PLAYING`), plus use fluent setter methods to define additional characteristics of the status, such as the current seek position.

The `DemoRouteController` logic for `ACTION_PLAY`, in the `onPlayRequest()` method, logs the event to LogCat and crafts a valid-but-meaningless result `Bundle` for use with `onResult()`:

```
  private boolean onPlayRequest(Intent i, ControlRequestCallback cb) {
    Log.d(getClass().getSimpleName(), "play: "
        + i.getData().toString());

    MediaItemStatus.Builder statusBuilder=
        new MediaItemStatus.Builder(
                                    MediaItemStatus.PLAYBACK_STATE_PLAYING);

    Bundle b=new Bundle();

    b.putString(MediaControlIntent.EXTRA_SESSION_ID,
DemoRouteProvider.DEMO_SESSION_ID);
    b.putString(MediaControlIntent.EXTRA_ITEM_ID,
DemoRouteProvider.DEMO_ITEM_ID);
    b.putBundle(MediaControlIntent.EXTRA_ITEM_STATUS,
                statusBuilder.build().asBundle());

    cb.onResult(b);

    return(true);
  }
```

Pause, Resume, and Stop

The `Bundle` passed to `onResult()` of the `ControlRequestCallback`, when the action is `ACTION_PAUSE`, `ACTION_RESUME`, or `ACTION_STOP`, does not need any particular values at the present time. Hence, the `DemoRouteController` methods for those actions just log the event to LogCat and pass an empty `Bundle` to `onResult()`:

```
  private boolean onPauseRequest(Intent i, ControlRequestCallback cb) {
    Log.d(getClass().getSimpleName(), "pause");

    cb.onResult(new Bundle());
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(true);
  }

  private boolean onResumeRequest(Intent i, ControlRequestCallback cb) {
    Log.d(getClass().getSimpleName(), "resume");

    cb.onResult(new Bundle());

    return(true);
  }

  private boolean onStopRequest(Intent i, ControlRequestCallback cb) {
    Log.d(getClass().getSimpleName(), "stop");

    cb.onResult(new Bundle());

    return(true);
  }
```

Get Status and Seek

The `Bundle` passed to `onResult()` of the `ControlRequestCallback`, when the action is `ACTION_GET_STATUS` or `ACTION_SEEK`, must contain the same sort of `MediaItemStatus`-built nested `Bundle` representing the current status. For `ACTION_GET_STATUS`, the only "work" to be done is to pass back the status; for `ACTION_SEEK`, you should move the playback position to the location indicated by an extra on the `Intent`, then return the revised status.

In the case of `DemoRouteController`, both just log a message to LogCat and return a fairly pointless status:

```
  private boolean onGetStatusRequest(Intent i,
                                     ControlRequestCallback cb) {
    Log.d(getClass().getSimpleName(), "get-status");

    MediaItemStatus.Builder statusBuilder=
        new MediaItemStatus.Builder(
                                     MediaItemStatus.PLAYBACK_STATE_PLAYING);

    Bundle b=new Bundle();

    b.putBundle(MediaControlIntent.EXTRA_ITEM_STATUS,
                statusBuilder.build().asBundle());

    cb.onResult(b);

    return(true);
  }

  private boolean onSeekRequest(Intent i, ControlRequestCallback cb) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    Log.d(getClass().getSimpleName(), "seek");

    MediaItemStatus.Builder statusBuilder=
        new MediaItemStatus.Builder(
                                    MediaItemStatus.PLAYBACK_STATE_PLAYING);

    Bundle b=new Bundle();

    b.putBundle(MediaControlIntent.EXTRA_ITEM_STATUS,
                statusBuilder.build().asBundle());

    cb.onResult(b);

    return(true);
  }
```

Publishing the Controller

While we have defined our `RouteController`, we still need to teach our `MediaRouteProvider` about it. That is through overriding the `onCreateRouteController()` method and returning an instance of `RouteController`:

```
  @Override
  public RouteController onCreateRouteController(String routeId) {
    return(new DemoRouteController());
  }
```

`onCreateRouteController()` is passed the route ID `String` used in the `MediaRouteDescriptor`. You can either use that to instantiate a different `RouteProvider`, pass the `String` into a common `RouteProvider` so it knows what to do, or ignore it entirely if you have only one published route. In the case of `DemoRouteProvider`, we ignore the route ID and always return a `DemoRouteController`.

Handling Discovery Requests

`DemoRouteProvider` is always available, largely because it does not do much of anything.

In the real world, your `MediaRouteProvider` may not always be relevant. For example, the TV you are set up to talk to may be powered down. Or, the user may not be at home where the TV is, so the client device and the TV are not on the same network.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Rather than constantly polling the outside world to see if a route is possible, we only do this when a client app requests "route discovery", such as by providing the `MediaRouter.CALLBACK_FLAG_REQUEST_DISCOVERY` flag on an `addCallback()` call to a `MediaRouter`. That in turn triggers an `onDiscoveryRequestChanged()` call on our `MediaRouteProvider`.

There, and in our constructor-triggered setup, we should do work to determine if a route is currently possible and set up our descriptors. This work should be done in a background thread if it involves network I/O.

Note that `onDiscoveryRequestChanged()` is passed a `MediaRouteDiscoveryRequest` object, describing what the consuming app is looking for. If the request is irrelevant for your provider (e.g., the app wants a local audio route, and you provide remote playback routes), simply ignore it.

The `onDiscoveryRequestChanged()` implementation in `DemoRouteProvider` just calls the same `handleDiscovery()` method that the constructor does.

Consuming the MediaRouteProvider

Having a `MediaRouteProvider` is nice, but it is useless if apps are not going to know about it.

You have two main options for consuming the `MediaRouteProvider`: use it only within your own app, or publish it to all apps on the device.

Private Provider

Using a `MediaRouteProvider` for your own app is very simple. Just add a single call to `addProvider()` on your `MediaRouter`, supplying an instance of your `MediaRouteProvider`.

Since our sample project is a fork of the original `RemotePlaybackClient` sample, we still have a `PlaybackFragment` that sets up the `MediaRouter` and `MediaRouteActionProvider`. In `onAttach()` of that `PlaybackFragment`, we can configure our `MediaRouterProvider` after obtaining the `MediaRouter` instance:

```
@Override
public void onAttach(Activity host) {
  super.onAttach(host);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    router=MediaRouter.getInstance(host);
    provider=new DemoRouteProvider(getActivity());
    router.addProvider(provider);
  }
```

At this point, our `DemoRouteProvider` will be available as an option for the user, along with any other eligible media routes:

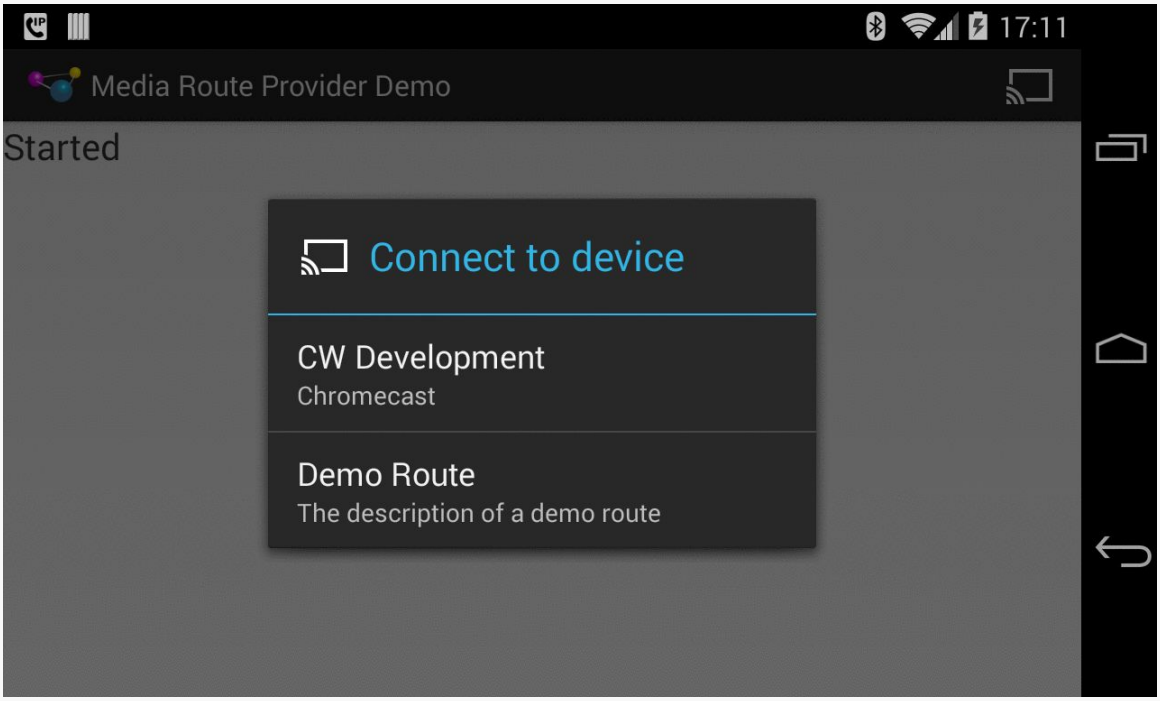


Figure 650: MediaRouteProvider Demo, on a Nexus 4, Showing Available Routes

Choosing the `DemoRouteProvider` ("Demo Route" in the screenshot) will allow you to use it just like you do a Chromecast... if you do not mind the fact that nothing shows up on your television:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

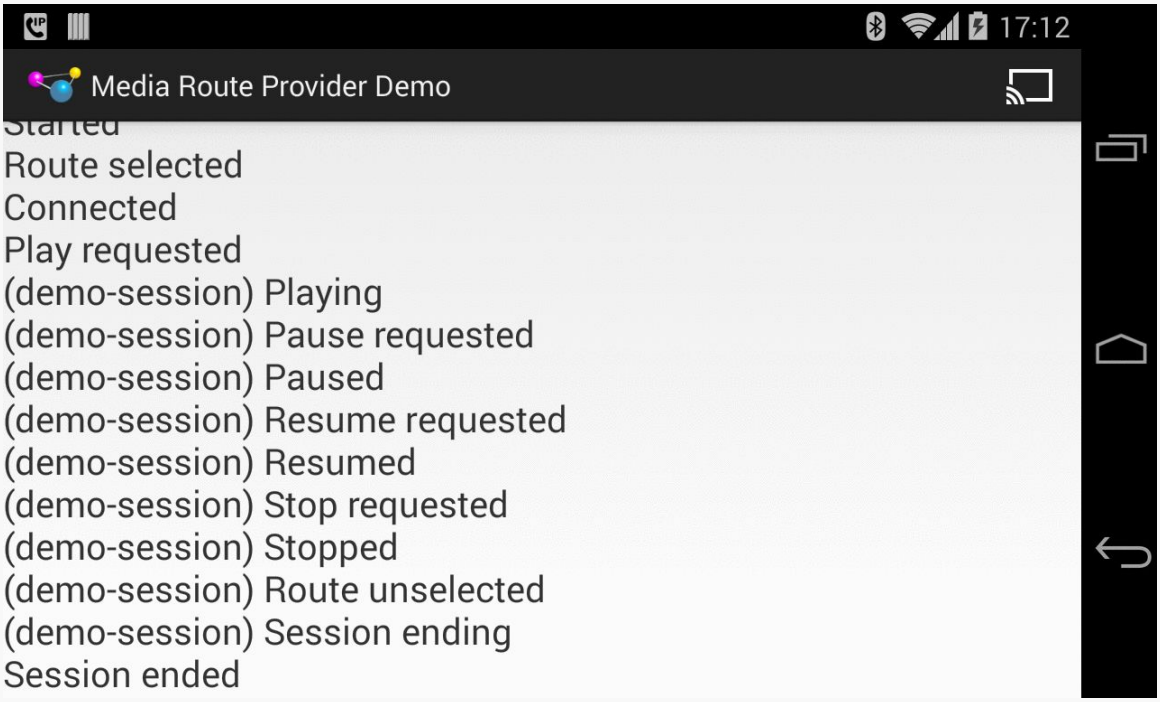


Figure 651: MediaRouteProvider Demo, on a Nexus 4, After Several Commands

As it turns out, the `DemoRouteProvider` works better than Google's own `MediaRouteProvider` for the Chromecast, insofar as more of the callbacks work. Specifically, we actually receive callbacks for pause, resume, and stop events, as opposed to having to just assume that those events completed.

Also, we remove the demo provider in `onDetach()`:

```
@Override
public void onDetach() {
  router.removeProvider(provider);

  super.onDetach();
}
```

Among other things, this allows us to correctly handle configuration changes — if we fail to call `removeProvider()` and blindly add another provider in `onAttach()`, we wind up with multiple providers, because our `MediaRouter` is a framework-provided singleton and is not re-created with the new fragment.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Public Provider

If you want your `MediaRouteProvider` to be used by other apps, you will need to create one more Java class: a subclass of `MediaRouteProviderService`. This requires only one method, `onCreateMediaRouteProvider()`, where you return an instance of your `MediaRouteProvider`:

```
package com.commonsware.android.mrp;

import android.support.v7.media.MediaRouteProvider;
import android.support.v7.media.MediaRouteProviderService;

public class DemoRouteProviderService extends MediaRouteProviderService {
  @Override
  public MediaRouteProvider onCreateMediaRouteProvider() {
    return(new DemoRouteProvider(this));
  }
}
```

This also needs to be added to your manifest, like any other `Service`. Give it an [intent-filter] looking for the `android.media.MediaRouteProviderService` action, so the media routing framework knows that it can obtain a `MediaRouteProvider` from it:

```
<service
  android:name="DemoRouteProviderService"
  tools:ignore="ExportedService">
  <intent-filter>
    <action android:name="android.media.MediaRouteProviderService"/>
  </intent-filter>
</service>
```

However, do *not* do both `addProvider()` *and* have the `<service>` element. If you use the `<service>` element, your app can use the `MediaRouteProvider`, just as can any other app on the device. Hence, in the published source code for this sample, the `<service>` element is commented out — you will need to uncomment it, and comment out the `addProvider()` call, to test the `DemoRouteProvider` with other apps.

Implementing This “For Realz”

Of course, `DemoRouteProvider` is just a demo and does not actually play any media anywhere. It is here to give you the basic steps for responding to `RemotePlaybackClient` requests. For a production `MediaRouteProvider`, in addition to the usual tightening-up of the code (e.g., better exception handling), you will

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

need to work on other areas as well, ones that are beyond the scope of the sample app.

Communicating with the Playback Device

Of course, the big one is passing the actions over to the playback device, so you actually *do* play back media.

If you are the developer of the playback device and its protocols (e.g., it is an Android device, and you are writing the playback app for it), then you can choose how you wish to handle the communications. You can work with low-level socket protocols directly, or you can leverage libraries like AllJoyn or ZeroMQ.

If the playback device "is what it is", and you cannot change it, then you will need to determine what protocols it offers and how best to map the `MediaControlIntent` actions to that protocol.

Also note that `onControlRequest()` is designed for asynchronous operation. The sample app just invoked the `ControlRequestCallback` during the `onControlRequest()` processing. Usually, though, your communications with the playback device will not be as fast as a call to `Log.d()`. You should arrange to do those communications in a background thread, perhaps via a single-thread thread pool as an `ExecutorService`. Simply pass the `ControlRequestCallback` to that thread along with the rest of the action's data (e.g., the URL of the media to load), and the thread can call `onResult()` or `onError()` as needed.

Handling Other Actions/Protocols

As was noted in the description of the sample app, that app eschews:

- volume control
- session management
- queue management

Any of those may be of interest to your users, and so you may need to consider offering them at some point. Also note that some potential client apps might need those capabilities and therefore will not see or use your published media routes without them.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Custom Actions

When setting up the `MediaRouteProvider`, we create one or more `MediaRouteDescriptor` objects wrapped around one or more `IntentFilter` objects. Those `IntentFilter` objects indicate what actions we support. The `DemoRouteProvider` uses standard actions (e.g., `ACTION_PLAY`) in a standard category (`CATEGORY_REMOTE_PLAYBACK`).

However, you are not limited to that.

You are welcome to also support custom actions in a custom category, to represent other things that your particular `MediaRouteProvider` offers. You can then use those actions from your own client app, or document them for use by third-party apps.

The client app can use `supportsControlRequest()` and `sendControlRequest()` to determine whether a particular media route supports a particular `Intent` that represents an action to be performed by that route's `MediaRouteProvider`. This way, a client app can work both with your custom `MediaRouteProvider` (taking advantage of your custom actions) and with regular providers that lack such support, assuming that the client can gracefully degrade its functionality.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Security

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

SSL

The traditional approach to securing HTTP operations is by means of SSL. Android supports SSL, much as ordinary Java does. Most of the time, you can just allow Android to do its thing with respect to SSL, and you will be fine. However, there may be times when you have to play a more direct role in SSL communications, to handle arbitrary SSL-encrypted endpoints, or to help ensure that your app is not the victim of a man-in-the-middle attack.

This chapter will explore various SSL scenarios and how to address them.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the chapter on Internet access.

Basic SSL Operation

Generally speaking, SSL "just works", for ordinary sites with ordinary certificates.

If you use an `https:` URL with `HttpUrlConnection`, `HttpClient`, or `WebView`, SSL handshaking will happen automatically, and assuming the certificates check out OK, you will get your result, just as if you had requested an `http:` URL.

However, `DownloadManager` only recently added support for SSL. Originally, requesting a download via `DownloadManager` with an `https:` scheme would result in `java.lang.IllegalArgumentException: Can only download HTTP URIs`. As of Android 4.0, SSL is supported. Hence, you need to be careful about making SSL

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

requests via `DownloadManager` to ensure that you are only doing that on a relatively recent version of Android.

For example, the Retrofit and Picasso sample apps from the chapter on Internet access both use `https://api.stackexchange.com` for their service endpoint. As a result, those requests — for the API JSON, at least — will go over SSL. You would need to log the URLs used for the image avatars to see whether StackExchange gives you `https` URLs or not.

Common SSL Problems

It would be nice if SSL always worked, rather than only generally worked. Alas, there are problems that you may well encounter while setting up SSL support, beyond having to deal with possible man-in-the-middle attacks.

This section will outline some of these common problems.

Self-Signed Certificate

SSL certificates used for public Web sites are usually backed by a "root certificate authority" that is well-known. That is not always the case.

One case is when the certificate is "self-signed", meaning that it was generated by somebody without involving a certificate authority. If you have shipped a production Android app, you created a self-signed certificate when you created your production key store. And you have been using a system-generated self-signed certificate throughout your development, known as the "debug signing key".

Self-signed certificates are rarely used on public-facing Web sites, as Web browsers are taught to warn users when such certificates are encountered. However, self-signed certificates might be used on internal servers, particularly test servers and other non-production environments.

There are even some benefits for using a self-signed certificate for production servers, if those servers will be talking only to your own apps and not arbitrary Web browsers. This technique will be reviewed later in this chapter.

Wildcard Certificate

Some certificates are difficult to validate because they use wildcards.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For example, Amazon S3 is a file storage and serving "cloud" solution from Amazon.com. They allow you to define "buckets" containing "objects", where each object then has its own URL. That URL is based on the name of the bucket and the name of the object. One option is for you to have the domain name of the URL be based on the name of the bucket, leaving the path to be solely the name of the object. This works, even with SSL, but Amazon needed to use a "wildcard SSL certificate", one that matches `*.s3.amazonaws.com`, not just a single domain name. By default, this will fail on Android, as Android's stock `TrustManager` will not validate wildcards for multiple domain name segments (e.g., `http://misc.commonsware.com.s3.amazonaws.com/foo.txt`). You will get an exception akin to:

```
javax.net.ssl.SSLHandshakeException: java.security.cert.CertificateException:
No subject alternative DNS name matching misc.commonsware.com.s3.amazonaws.com
found
```

Custom Certificate Authority

Some larger organizations have set up their own certificate authority. Sometimes, they aspire to become a recognized root certificate authority, but have not been adopted by many browsers. Sometimes, they simply want to have more structure than a pure self-signed certificate but do not necessarily want to have all certificates go through a root certificate authority, perhaps due to expense.

In these cases, Android will reject the SSL certificate, for the same reason it rejects self-signed ones: it cannot validate the certificate chain all the way back to a known root certificate authority. But, with a little work, you can enable Android to support these as well.

TrustManagers

Our work in handling these common problems — and some other enhancements that help with other sorts of attacks — revolves around a `TrustManager`. You may already be familiar with these from standard Java development. Otherwise, this section reviews what a `TrustManager` is and how you apply one in your Android HTTP operations. Later on, we will discuss where you can get a `TrustManager` that works the way *you* want.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What is a TrustManager?

In Java, a `TrustManager` is responsible for determining if an SSL certificate chain is valid or not for a given request.

An SSL certificate chain, in turn, represents a series of SSL certificates that, as the name suggests, are chained together. Sometimes, the chain will only have one entry (e.g., a self-signed certificate). More often, the chain will have a few entries, starting with the server's certificate, which may chain to some intermediate certificate(s), eventually leading to a certificate that might be from a root certificate authority.

In truth, we often use a collection of `TrustManager` instances, represented as a simple Java array. And, of course, courtesy of design patterns like the composite pattern, one `TrustManager` might delegate decision-making to other `TrustManager` instances.

How Do We Use One?

For the moment, let us assume that we get a `TrustManager` array handed to us by magic. How would we use it?

The answer depends on what you are using for your HTTP stack, as is outlined below.

HttpsURLConnection

Normally, when using a `java.net.URL`, you open a connection and do not pay much attention to whether the connection object you get back is an `HttpURLConnection` or an `HttpsURLConnection`. If, however, you want to tailor the SSL processing with a custom `TrustManager` array, you will need to explicitly use `HttpsURLConnection`:

```
SSLContext ssl=SSLContext.getInstance("TLS")

ssl.init(null, builders, null);

HttpsURLConnection conn=(HttpsURLConnection)new URL(url).openConnection();

conn.setSSLSocketFactory(ssl.getSocketFactory());

// conn.getInputStream() and other work to process the HTTPS call
```

(here, `builders` is the array of `TrustManager` instances)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

OkHTTP

Square's OkHttp works along the same lines:

```
SSLContext ssl=SSLContext.getInstance("TLS")

ssl.init(null, builders, null);

OkHttpClient okHttpClient=new OkHttpClient();

okHttpClient.setSslSocketFactory(ssl.getSocketFactory());

HttpURLConnection conn=okHttpClient.open(new URL(url));

// conn.getInputStream() and other work to process the HTTPS call
```

Something Else

Other HTTP stacks, or libraries that in turn use HTTP directly or indirectly, should offer similar facilities. If not, you may wish to switch to something that does.

What Not To Do

You will find various blog posts, Stack Overflow answers, and the like that suggest that you simply disable SSL certificate validation, by implementing an "accept-all" `TrustManager`. Such a `TrustManager` basically implements the interface with empty stubs for methods like `checkServerTrusted()`, not throwing any exceptions.

Technically, this works. And, *if* you are using this only early on in development and *if* you swear upon a stack of `$RELIGIOUS_TEXTS` that you will replace this hack by the time you go to production, it is difficult to complain about this technique.

However, in production, ignoring SSL certificate validation errors opens your app up to man-in-the-middle attacks. We will explore man-in-the-middle attacks in greater detail later in this chapter.

If you think that this is not important, bear in mind that the US Federal Trade Commission (FTC) has sued firms for blindly accepting all SSL certificates. It is not out of the question for class action suits, or legal action by other nations, to be forthcoming.

Please do not implement an "accept-all" `TrustManager`!

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

TrustManagerBuilder

To simplify handling some of these scenarios, the author of this book has written a `TrustManagerBuilder`, as part of the CWAC-Security library. The objective of this library is to bundle up some good SSL hygiene practices in an easy-to-use builder-style API. You create an instance of `TrustManagerBuilder`, call a series of methods on it to configure your desired SSL behavior, then call `buildArray()` to get the `TrustManager` array to pass into your `init()` call on the `SSLSocket`, as described above.

The following sections are based in part on the `TrustManagerBuilder` documentation.

Basic Configuration

There are two `TrustManagerBuilder` constructors: a zero-argument constructor and a one-parameter constructor, taking a `Context`. Use the one-parameter constructor if you plan on using the builder-style methods that take a raw resource ID or a path into `assets/` as parameters.

Of course, the real work is done in the `// configure builder here` parts, using the following builder-style methods:

- `useDefault()`: this tells `TrustManagerBuilder` to use the system default `TrustManager` for validating incoming SSL certificates
- `selfSigned()`: this tells `TrustManagerBuilder` to allow a specific self-signed SSL certificate, based upon a supplied keystore file and password
- `allowCA()`: this tells `TrustManagerBuilder` to accept certificates signed by a custom certificate authority, based upon a supplied certificate file
- `denyAll()`: this tells `TrustManagerBuilder` to reject all certificates (mostly for testing purposes)
- `memorize()`: this tells `TrustManagerBuilder` to only accept SSL certificates approved by the user
- `addAll()`: this tells `TrustManagerBuilder` to add other trust managers that you may have implemented yourself or obtained from other libraries

In addition, `or()` tells `TrustManagerBuilder` to logically OR any subsequent configuration with whatever came previously in the build, while `and()` indicates that subsequent configuration should be logically AND-ed with whatever came previously.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Supporting Self-Signed Certificates

Perhaps you are using self-signed certificates for an in-house server, such as a test server for development purposes.

`selfSigned()` will help with that. The simple form of the method takes two parameters. The first parameter is either:

- a `File` pointing to a keystore for your self-signed certificate on the local file system,
- an `int` raw resource ID, if you wish to package the keystore in your app in `res/raw/`, or
- a `String` pointing to a relative path in `assets/` where you have placed your keystore

The second parameter is a `char` array for the password for the keystore. If you are dynamically retrieving that password (e.g., the user types it in), you can clear out the `char` array (e.g., set all elements in the array to `x`), to quickly get rid of the password from memory. If your password is more static, just call `toCharArray()` on a `String` to get the `char` array that you need.

The default `selfSigned()` expect that keystore to be in BKS format; if your keystore is in some other format supported by Android's edition of `KeyStore`, use the three-parameter version of `selfSigned()` that takes the `KeyStore` format name as the last parameter.

If you want to *only* support a specific self-signed certificate, you could set up a `TrustManagerBuilder` as follows:

```
new TrustManagerBuilder(this).selfSigned(R.raw.selfsigned,
"foobar".toCharArray());
```

(where `this` is a `Context`, like the `IntentService` in which you are making a SSL-encrypted Web service call)

Here, the BKS-formatted keystore would reside in `res/raw/selfsigned.bks` (though the file extension could vary).

If you want to support more than one self-signed certificate — such as when you plan on switching your old certificate to a new one — you could do:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
new TrustManagerBuilder(this)
  .selfSigned(R.raw.selfsigned, "foobar".toCharArray())
  .or()
  .selfSigned(R.raw.selfsigned2, "snicklefritz".toCharArray());
```

If you want to use a single `TrustManagerBuilder` for both your self-signed scenario and regular certificate authority-based certificates, you could do:

```
new TrustManagerBuilder(this)
  .selfSigned(R.raw.selfsigned, "foobar".toCharArray())
  .or()
  .useDefault();
```

Getting the self-signed certificate keystore, and employing the self-signed certificate on your server, is up to you. For example, you could use `openssl` to generate a 4,096-bit self-signed certificate, with a lifespan of 10,000 days:

```
openssl req -new -newkey rsa:4096 -sha1 -days 10000 -nodes -x509 -keyout
selfsigned.key -out selfsigned.crt
```

(note: this should all be on one line; it is split over multiple lines here to fit the page)

That will give you something that you could use with your server. To get a BKS file for use with `TrustManagerBuilder`, you can use the JAR for The Legion of the Bouncy Castle in conjunction with the `keytool` command. For example, the following command will take the self-signed certificate from above and create `selfsigned.bks` as output, using `foobar` as the password, assuming that the `bcprov-jdk16-146.jar` JAR file is in the current working directory:

```
keytool -import -file selfsigned.crt -keystore selfsigned.bks -storetype BKS
-provider org.bouncycastle.jce.provider.BouncyCastleProvider -providerpath
bcprov-jdk16-146.jar -storepass foobar
```

(note: this should all be on one line; it is split over several lines here to fit the page)

Supporting Custom Certificate Authorities

The Android developer documentation has some sample code for validating an SSL certificate against a custom certificate authority (CA). That same technique is integrated into `TrustManagerBuilder`.

```
new TrustManagerBuilder(getContext()).allowCA("uwash-load-der.crt")
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here, `uwash-load-der.crt` is the CA's root certificate (in this case, from the University of Washington). The parameter to `allowCA()` is similar to the parameter to `selfSigned()`:

- a `File` pointing to the CA's root certificate on the local file system,
- an `int` raw resource ID, if you wish to package the CA's root certificate in your app in `res/raw/`, or
- a `String` pointing to a relative path in `assets/` where you have placed your CA's root certificate

By default, the one-parameter version of `allowCA()` assumes an X.509 certificate file. If your certificate file is in some other format that is supported by Android's edition of the `CertificateFactory` class, you can use the two-parameter version of `allowCA()` that takes the format name as a `String` in the second parameter.

And, of course, `allowCA()` can be combined with the others as well, such as a configuration that supports the default certificate authorities or a custom one:

```
new TrustManagerBuilder(this)
  .allowCA("uwash-load-der.crt")
  .or()
  .useDefault();
```

About That Man in the Middle

Man-in-the-middle (MITM) attacks are a common way of trying to intercept SSL encrypted communications. The "man" in the "middle" might be a proxy server, a different Web site you wind up communicating with via DNS poisoning, etc. The objective of the "man" is to pretend to be the actual Web site or Web service you are trying to communicate with. If your app "falls for it", your app will open an encrypted channel to the attacker, not your site, and the attacker will have access to the unencrypted data you send over that channel.

Unfortunately, Android apps have a long history of being victims of man-in-the-middle attacks.

Fahl, Harbach, Muders, and Smith of the Leibniz University of Hannover, and Baumgärtner and Freisleben of the Philipps University of Marburg, conducted research on this problem. Their results, as reported in their paper, "Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security", are depressing. One in six surveyed apps explicitly ignored SSL certificate validation issues, mostly

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

by means of do-nothing `TrustManager` implementations as noted above. Out of a selected 100 apps, 41 could be successfully attacked using man-in-the-middle techniques, yielding a treasure trove of credit card information, account credentials for all the major social networks, and so forth.

Their paper outlines a few ways in which apps can screw up SSL management — the following sections outline some of them.

Disabling SSL Certificate Validation

As mentioned above, if you disable SSL certificate validation, by implementing and using a do-nothing `TrustManager`, you are wide open for man-in-the-middle attacks. A simple transparent proxy server can pretend to be the real endpoint — apps ignoring SSL validation entirely will trust that the transparent proxy is the real endpoint and, therefore, perform SSL key exchange with the proxy rather than the real site. The proxy, as a result, gets access to everything the app sends.

Ignoring Domain Names

A related flaw is when you disable hostname verification. The "common name" (CN) of the SSL certificate should reflect the domain name being requested. Requesting `https://www.foo.com/something` and receiving an SSL certificate for `xkcdhatguy.com` would be indicative of a mis-configured Web server at best and a man-in-the-middle attack at worst.

By default, this is checked, and if there is no match, you will get errors like:

```
javax.net.ssl.SSLException: hostname in certificate didn't match: <...>
```

where the `...` is replaced by whatever domain name you were requesting.

But some developers disable this check. Perhaps during development they were accessing the server using a private IP address, and they were getting `SSLExceptions` when trying to access that server. It is very important to allow Android to check the hostname for you, which is the default behavior.

Note that HttpClient's `BrowserCompatHostnameVerifier` — which applies standard Web browser rules for handling hostname verification — is broken on Android 2.1 and earlier, failing to handle wildcard certificates properly.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Hacked CAs

The truly scary issue is when the problem stems from the CA itself.

Comodo, TURKTRUST, and other certificate authorities have been hacked, where nefarious parties gained the ability to create arbitrary certificates backed by the CA. For example, in the TURKTRUST case, Google found that somebody had created a `*.google.com` certificate that had TURKTRUST as the root CA. Any browser — or Android app — that implicitly trusted TURKTRUST-issued certificates would believe that this certificate was genuine. This is the ultimate in man-in-the-middle attacks, as code that is ordinarily fairly well-written will believe the CA and therefore happily communicate with the attacker.

Self-Signed Certificates, Revisited

As Moxie Marlinspike points out, one way to avoid having your app be the victim of a man-in-the-middle attack due to a hijacked certificate authority is to simply *not use a certificate authority*.

Certificate authorities are designed for use by general-purpose clients (e.g., Web browsers) hitting general-purpose servers (e.g., Web servers). In the case where you control both the client *and* the server, you don't need a certificate authority. You merely need to have a self-signed certificate that both ends know about.

This works well if the Web server is solely functioning as a Web service to deliver data to your Android app, or perhaps other native apps on other platforms for which you can also support self-signed certificates. Depending upon your server's capabilities, you might be able to arrange to have the same server-side application logic be available both from a self-signed certificate on one domain (for use with apps) and from a CA-rooted certificate for another domain (for use with Web browsers).

However, it is very possible that the staff who manage the servers will reject the notion of using a self-signed certificate, perhaps in an effort to minimize the complexity of supporting multiple SSL paths (for browsers and apps). Or, you may not control the server well enough to go with a self-signed certificate, such as if you are using a cloud computing provider.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Certificate Memorizing

If your app needs to connect to arbitrary SSL servers — perhaps ones configured by the user (e.g., email client) or are intrinsic to the app's usage (e.g., URLs in a Web browser) — detecting man-in-the-middle attacks boils down to proper SSL certificate validation... and praying for no hacked CA certificates.

However, one way to incrementally improve security is to use certificate memorizing. With this technique, each time you see a certificate that you have not seen before, or perhaps a different certificate for a site visited previously, you ask the user to confirm that it is OK to proceed.

The idea here is that even if we cannot tell, absolutely, whether a given certificate is genuine or from an attacker, we can detect *differences in certificates over time*. So, if the user has been seeing certificate A, and now all of a sudden receives certificate B instead, there are two main possibilities:

1. The HTTPS server changed certificates for legitimate reasons
2. An attacker is providing an alternative certificate

So, what we do is check certificates against a roster that the user has approved before. If the newly-received certificate is not in that roster, we fail the HTTPS request, but raise a custom exception so that your code can detect this case and ask the user for approval to proceed.

Technically savvy users may be able to deduce whether the certificate is indeed genuine; slightly less-savvy users might simply contact the site to see if this is expected behavior. The downside is that technically unsophisticated users might be baffled by the question of whether or not they should accept the certificate and may take their confusion out on you, the developer of the app that is asking the question.

The Standalone Solution

There is a standalone implementation of a `MemorizingTrustManager` that you could consider using. It has been around for a few years, with a slow-but-steady set of updates.

However, that library handles asking the user for acceptance of the certificates for you, rather than raising some event that your app can handle itself. In order to tailor the UI, you would need to modify the library itself.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Moreover, the library attempts to handle this UI *while your SSL request is in process*, by blocking the background thread upon which you are making the HTTPS request. A side-effect of this is that `MemorizingTrustManager` has some fairly unpleasant code for trying to block this thread while interacting with the user on the main application thread. And, if the user takes too long, your request to the server may time out anyway.

The TrustManagerBuilder Solution

The author of this book has created a separate `MemorizingTrustManager` that is part of the `TrustManagerBuilder` of the CWAC-Security library. It will simply throw a custom exception if a certificate is not memorized. Your code that request the HTTPS operation is already needing to catch various exceptions. If your code detects this special `CertificateNotMemorizedException`, you can handle asking the user what to do yourself, integrated into your own UI, without threading concerns. If the user elects to proceed, you can have `TrustManagerBuilder` memorize the certificate, after which you can try your HTTPS request again.

This requires a bit more configuration and management than much of the rest of `TrustManagerBuilder`. There is a dedicated demo project in the `demoMemo/` directory of the CWAC-Security repository that demonstrates how to use certificate memorization. The sections that follow explain how to set up certificate memorization, and are based upon the `TrustManagerBuilder` documentation.

Configuring the TrustManagerBuilder

In addition to other builder methods, you can call `memorize()` on a `TrustManagerBuilder` to indicate that you want to enable certificate memorization. This method takes an instance of a `MemorizingTrustManager.Options` object to configure how memorization works.

The constructor for `MemorizingTrustManager.Options` takes three parameters:

1. A `Context`, used only for the duration of the constructor itself — this `Context` is not retained after the constructor returns.
2. A `String` representing a relative path to a directory, inside of `getFilesDir()`, for working files for certificate memorization. This directory will be created for you if it does not already exist.
3. A `String` that is the password to use for the `KeyStore` that will hold the memorized certificates.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

While `MemorizingTrustManager.Options` offers a builder-style API for configuring the options, no other methods are required beyond the constructor for basic use.

The `memorize()` call should be towards the end of the configuration, after an `and()` call, to say "we will accept SSL certificates that match other criteria *and* are ones that are memorized".

```
options=
    new MemorizingTrustManager.Options(this, "memorize", "snicklefritz");

try {
  builder=
      new TrustManagerBuilder(this).useDefault().and().memorize(options);
}
catch (Exception e) {
  // do something useful
}
```

Detecting Memorization Exceptions

When you perform an HTTPS operation, you may get an `SSLHandshakeException`. That should wrap another exception, retrieved via `getCause()`.

If the wrapped exception is `CertificateNotMemorizedException`, that means that we found a certificate that matched your other criteria (e.g., valid root CA), but was not found in the memorized roster of certificates. The `CertificateNotMemorizedException` contains the certificate chain, retrieved via `getCertificateChain()`.

At this point, you should tell the user that the request failed because of the unrecognized certificate, and ask the user how to proceed. You are welcome to use the contents of the certificate chain to provide technical details regarding the unrecognized certificates to the user, if you so choose.

Bear in mind that your users may be non-technical. Throwing up a dialog with a lot of SSL gibberish may not be effective. Instead, ideally, you should steer them for how to contact somebody who can indicate if it is safe to proceed.

There are three possible avenues that the user could take:

1. The user could elect to not proceed, under the premise that the SSL communications may be compromised. How you handle that is up to you.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. The user could elect to proceed, but not remember this certificate for very long ("Allow once").
3. The user could elect to proceed, with you remembering the certificate for a long time, if the certificate is thought to be valid.

Memorizing the Certificate

In cases #2 and #3 above, you can call methods on your `TrustManagerBuilder` to update the memorized roster with the certificate chain that had failed previously. Use `allowCertOnce()` to remember the certificate for the lifetime of your process, and use `memorizeCert()` to remember the certificate indefinitely. Both of these methods take the `X509Certificate` array that is the certificate chain that you got from calling `getCertificateChain()` on the `CertificateNotMemorizedException`.

After you do this, you can re-try your request that failed due to the unrecognized certificate, and it should succeed.

Clearing the Certificate Roster

At any point, you can call `clearMemorizedCerts()` on the `TrustManagerBuilder` to get rid of the certificate roster. Pass `false` as the parameter to only get rid of the "transient" certificates (i.e., the ones you registered via `allowCertOnce()`). Pass `true` as the parameter to get rid of both the transient and the persistent certificates (i.e., the ones you registered via `memorizeCert()`).

Notes on Threading

Because certificate memorization involves reading from and writing to files, setup and use of the `TrustManagerBuilder` should be performed on a background thread. Of course, your network I/O should be on a background thread, anyway.

If you will have several threads that are all performing HTTPS operations, they should *share* a `TrustManagerBuilder` instance, so that there is a central spot for updating the certificate roster. The `MemorizingTrustManager` should be thread-safe; please file issues if you run into threading-related problems.

Using Trust-on-First-Use

The default behavior of certificate memorization is to fail on every unrecognized certificate. The downside of this is that the user will immediately get a failure the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

first time the user uses the app, because no certificates will have been memorized by that point.

You have three courses of action to handle this:

1. Live with it.
2. Manage it yourself, by determining when you believe it is safe to memorize the certificate without user involvement.
3. Call `trustOnFirstUse()` on the `MemorizingTrustManager.Options` object when you create it, to indicate that the *first* unrecognized certificate should be memorized automatically, with failures being reported for all subsequent unrecognized certificates.

About CertificateMemorizationException

If you encounter a `CertificateMemorizationException` — in a crash log, for example — that indicates that the library encountered some problem when attempting to save a certificate that is being saved automatically via the trust-on-first-use feature. This exception is unlikely to occur.

Pinning

One way to limit the possible damage from hacked CAs is to recognize that most apps do not need to communicate with arbitrary servers. Web browsers, email clients, chat clients, and the like might need to be able to communicate with whatever server the user elects to configure. But many apps just need to communicate back to their developer's own server, such as a native client adjunct to a regular Web app.

In this case, the app does not need to accept arbitrary SSL certificates. The developer knows the actual SSL certificate used by the developer's server, so the developer can arrange to accept only that one certificate. Or, the developer knows the CA that they get their SSL certificates from and can only accept certificates issued by that CA, and not other CAs. This reduces security a bit, but makes it easier for you to handle SSL certificate expiration and replacement without major headaches.

This technique is referred to as "pinning". Chrome is perhaps the most well-known implementer of pinning: when you access services like Gmail from Chrome, Google (who wrote the browser) knows the valid certificates for Google (who wrote the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

server) and can toss out anything that is invalid... such as the TURKTRUST fake certificate mentioned earlier in this chapter.

Nikolay Elenkov has another blog post outlining certificate pinning support in Android 4.2+. Moxie Marlinspike has an implementation of pinning, with a description in a blog post, and has also released a pinning library offering a `PinningTrustManager` that you can use following his guidelines.

NetCipher

The Guardian Project has released an Android library project called NetCipher — formerly known as OnionKit — designed to help boost Internet security for Android applications.

In particular, NetCipher helps your application integrate with Orbot, a Tor proxy. Tor ("The Onion Router") is designed to help with anonymity, having your Internet requests go through a series of Tor routers before actually connecting to your targeted server through some Tor endpoint. Tor is used for everything from mitigating Web site tracking to helping dissidents bypass national firewalls. NetCipher helps your app:

- Detect if Orbot is installed, and help the user install it if it is not
- Detect if Orbot is running, and help you start it if it is not
- Make HTTP requests by means of Orbot instead of directly over the Internet

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Advanced Permissions

Adding basic permissions to your app to allow it to, say, access the Internet, is fairly easy. However, the full permissions system has many capabilities beyond simply asking the user to let you do something. This chapter explores other uses of permissions, from securing your own components to using signature-level permissions (your own or Android's).

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on permissions and the chapter on signing your app. The discussion of signature-level permissions will make a bit more sense if you read through the chapter on plugins as well.

Securing Yourself

Principally, at least initially, permissions are there to allow the user to secure their device. They have to agree to allow you to do certain things, such as reading contacts, that they might not appreciate.

The other side of the coin, of course, is to secure your own application. If your application is mostly activities, security may be just an "outbound" thing, where you request the right to use resources of other applications. If, on the other hand, you put content providers or services in your application, you will want to implement "inbound" security to control which applications can do what with the data.

Note that the issue here is less about whether other applications might "mess up" your data, but rather about privacy of the user's information or use of services that

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

might incur expense. That is where the stock permissions for built-in Android applications are focused – can you read or modify contacts, can you send SMS, etc. If your application does not store information that might be considered private, security is less an issue. If, on the other hand, your application stores private data, such as medical information, security is much more important.

The first step to securing your own application using permissions is to declare said permissions, once again in the `AndroidManifest.xml` file. In this case, instead of `uses-permission`, you add `permission` elements. Once again, you can have zero or more `permission` elements, all as direct children of the root `manifest` element.

Declaring a permission is slightly more complicated than using a permission. There are three pieces of information you need to supply:

- The symbolic name of the permission. To keep your permissions from colliding with those from other applications, you should use your application's Java namespace as a prefix
- A label for the permission: something short that would be understandable by users
- A description for the permission: something a wee bit longer that is understandable by your users

```
<permission
  android:name="vnd.tlagency.sekrits.SEE_SEKRITS"
  android:label="@string/see_sekrits_label"
  android:description="@string/see_sekrits_description" />
```

This does not enforce the permission. Rather, it indicates that it is a possible permission; your application must still flag security violations as they occur.

Enforcing Permissions via the Manifest

There are two ways for your application to enforce permissions, dictating where and under what circumstances they are required. The easier one is to indicate in the manifest where permissions are required.

Activities, services, and receivers can all declare an attribute named `android:permission`, whose value is the name of the permission that is required to access those items:

```
<activity
  android:name=".SekritApp"
  android:label="Top Sekrit"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:permission="vnd.tlagency.sekrits.SEE_SEKRITS">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category
      android:name="android.intent.category.LAUNCHER"
/>
  </intent-filter>
</activity>
```

Only applications that have requested your indicated permission will be able to access the secured component. In this case, "access" means:

1. Activities cannot be started without the permission
2. Services cannot be started, stopped, or bound to an activity without the permission
3. Intent receivers ignore messages sent via `sendBroadcast()` unless the sender has the permission

Enforcing Permissions Elsewhere

In your code, you have two additional ways to enforce permissions.

Your services can check permissions on a per-call basis via `checkCallingPermission()`. This returns `PERMISSION_GRANTED` or `PERMISSION_DENIED` depending on whether the caller has the permission you specified. For example, if your service implements separate read and write methods, you could require separate read versus write permissions in code by checking those methods for the permissions you need from Java.

Also, you can include a permission when you call `sendBroadcast()`. This means that eligible broadcast receivers must hold that permission; those without the permission are ineligible to receive it. We will examine `sendBroadcast()` in greater detail elsewhere in this book.

Requiring Standard System Permissions

While normally you require your own custom permissions using the techniques described above, there is nothing stopping you from reusing a standard system permission, if it would fit your needs.

For example, suppose that you are writing YATC (Yet Another Twitter Client). You decide that in addition to YATC having its own UI, you will design YATC to be a "Twitter engine" for use by third party apps:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Send timeline updates via broadcast `Intents`
- Publish the timeline, the user's own tweets, @-mentions, and the like via a `ContentProvider`
- Offer a command-based service interface for posting updates to the timeline
- And so on

You could, and perhaps should, implement your own custom permission. However, since any app can get to Twitter just by having the `INTERNET` permission, one could argue that a third-party app should just need that same `INTERNET` permission to use your API (rather than integrating JTwitter or another third-party JAR).

Signature Permissions

Each permission in Android is assigned a protection level, via an `android:protectionLevel` attribute on the `<permission>` element. By default, permissions are at a `normal` level, but they can also be flagged as `dangerous`, `signatureOrSystem`, or `signature`. In the latter two cases, "signature" means that the app requesting the permission and the app requiring the permission should have be signed by the same signing key. In the case of `signatureOrSystem` — only used by the firmware – the app requesting the permission either needs to be signed by the firmware's signing key or reside on the system partition (e.g., come pre-installed with the device).

Firmware-Only Permissions

Most of Android's permissions mentioned in this book are ones that any SDK application can hold, if they ask for them and the user grants them. `INTERNET`, `READ_CONTACTS`, `ACCESS_FINE_LOCATION`, and kin all are normal permissions.

`BRICK` is not.

There is a permission in Android, named `BRICK`, that, in theory, allows an application to render a phone inoperable (a.k.a., "brick" the phone). While there is no `brickMe()` method in the Android SDK tied to this permission, presumably there might be something deep in the firmware that is protected by this permission.

The `BRICK` permission cannot be held by ordinary Android SDK applications. You can request it all you want, and it will not be granted.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, applications that are signed with the same signing key that signed the firmware *can* hold the `BRICK` permission.

That is because the system's own manifest has the following `<permission>` element:

```
<permission android:name="android.permission.BRICK"
        android:label="@string/permlab_brick"
        android:description="@string/permdesc_brick"
        android:protectionLevel="signature" />
```

Some other permissions have `signatureOrSystem` instead of `signature` for `android:protectionLevel`:

```
<permission android:name="android.permission.REBOOT"
        android:label="@string/permlab_reboot"
        android:description="@string/permdesc_reboot"
        android:protectionLevel="signatureOrSystem" />
```

These permissions can be held by applications that are *either* signed by the firmware's signing key *or* by applications that are installed on the firmware's partition. Mostly, this will be apps that are licensed by a manufacturer or carrier for pre-distribution on a device.

Your Own Signature Permissions

You too can require `signature`-level permissions. That will restrict the holders of that permission to be other apps signed by your signing key. This is particularly useful for inter-process communication between apps in a suite — by using `signature` permissions, you ensure that only your apps will be able to participate in those communications.

This is what was used in the `ContentProvider`-based plugin sample from elsewhere in this book. The plugin required a permission that was declared with `android:protectionLevel="signature"`, and the host application requested that permission.

One nice thing about these sorts of signature-level permissions is that the user is not bothered with them. It is assumed that the user will agree to the communication between the apps signed by the same signing key. Hence, the user will not see signature-level permissions at install or upgrade time.

Since in some cases, you may not be sure which app will be installed first, it is best to have all apps in the suite include the *same* `<permission>` element, in addition to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the corresponding `<uses-permission>` element. That way, no matter which app is installed first, it can declare the permission that all will share.

Though, that has its own problems, as you will see in the next section.

The Custom Permission Vulnerability

NOTE: Some of the material in this section originally appeared in material hosted in the CWAC-Security project repository. In addition, the author would like to thank Mark Carter and "Justin Case" for their contributions in this topic area).

Unfortunately, custom permissions have some undocumented limitations that make them intrinsically risky. Specifically, custom permissions can be defined by anyone, at any time, and "first one in wins", which opens up the possibility of unexpected behavior.

Here, we will walk through some scenarios and show where the problems arise, plus discuss how to mitigate them as best we can.

Scenarios

All of the following scenarios focus on three major app profiles.

App A is an app that defines a custom permission in its manifest, such as:

```
<permission
  android:name="com.commonsware.cwac.security.demo.OMG"
  android:description="@string/perm_desc"
  android:label="@string/perm_label"
  android:protectionLevel="normal"/>
```

App A also defends a component using the `android:permission` attribute, referencing the custom permission:

```
<provider
  android:name="FileProvider"
  android:authorities="com.commonsware.cwac.security.demo.files"
  android:exported="true"
  android:grantUriPermissions="false"
  android:permission="com.commonsware.cwac.security.demo.OMG">
  <grant-uri-permission android:path="/test.pdf"/>
</provider>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

App B has a `<uses-permission>` element to declare to the user that it wishes to access components defended by that permission:

```
<uses-permission android:name="com.commonsware.cwac.security.demo.OMG"/>
```

App C has the same `<uses-permission>` element. The difference is that App B *also* has the `<permission>` element, just as App A does, albeit with different descriptive information (e.g., `android:description`) and, at times, a different protection level.

All three apps are signed with different signing keys, because in the real world they would be from different developers.

So, to recap:

- A defines a permission and uses it for defense
- B defines the same permission and requests to hold it
- C just requests to hold this permission

With all that in mind, let's walk through some possible scenarios, focusing on two questions:

1. What is the user told, when the app is installed through normal methods (i.e., not via `adb`), regarding this permission?
2. What access, if any, does App B or App C have to the `ContentProvider` from App A?

The Application SDK Case (A, Then C)

Suppose the reason why App A has defined a custom permission is because it wants third-party apps to have the ability to access its secured components... but only with user approval. By defining a custom permission, and having third-party apps request that permission, the user should be informed about the requested permission and can make an informed decision.

Conversely, if an app tries to access a secured component but has not requested the permission, the access attempt should fail.

App C has requested the custom permission via the `<uses-permission>` element. If the permission — defined by App A — has an `android:protectionLevel` of `normal` or `dangerous`, the user will be informed about the requested permission at install

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

time. If the user continues with the installation, App C can access the secured component.

If, however, the `android:protectionLevel` is `signature`, the user is not informed about the requested permission at install time, as the system can determine on its own whether or not the permission should be granted. In this case, App A and App C are signed with different signing keys, so Android silently ignores the permission request. If the user continues with installation, then App C tries to access App A's secured component, App C crashes with a `SecurityException`.

In other words, this all works as expected.

The Application SDK Problem Case (C, Then A)

However, in many cases, there is nothing forcing the user to install App A before App C. This is particularly true for publicly-distributed apps on common markets, like the Play Store.

When the user installs App C, the user is not informed about the request for the custom permission, presumably because that permission has not yet been defined. If the user later installs App A, App C is not retroactively granted the permission, and so App C's attempts to use the secured component fail.

This works as expected, though it puts a bit of a damper on custom permissions. One way to work around this would be for the user to uninstall App C, then install it again (with App A already installed). This returns us to the original scenario from the preceding section. However, if the user has data in App C, losing that data may be a problem (as in a "let's give App C, or perhaps App A, one-star ratings on the Play Store" sort of problem).

The Peer Apps Case, Part One (A, Then B)

Suppose now we augment our SDK-consuming app (formerly App C) to declare the same permission that App A does, in an attempt to allow the two apps to be installed in either order. That is what App B is: the same app as App C, but where it has the same `<permission>` element as does App A in its manifest.

This scenario is particularly important where both apps could be of roughly equal importance to the user. In cases where App C is some sort of plugin for App A, it is not unreasonable for the author of App A to require App A to be installed first. But,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

if Twitter and Facebook wanted to access components of each others' apps, it would be unreasonable for either of those firms to mandate that their app must be installed first. After all, if Twitter wants to be installed first, and Facebook wants to be installed first, one will be disappointed.

If the user installs App A (the app defending a component with the custom permission) before App B, the user will be notified at install time about App B's request for this permission. Notably, the information shown on the installation security screen will contain App A's description of the permission. And, if the user goes ahead and installs App B, App B can indeed access App A's secured component, since it was granted permission by the user.

Once again, everything is working as expected. Going back to the two questions:

1. The user is informed when App B or App C requests the permission defined by App A.
2. App B and App C can hold that permission if and only if they meet the requirements of the protection level

The Peer Apps Case, Part Two (B, Then A)

What happens if we reverse the order of installation? After all, if App A and App B are peers, from the standpoint of the user, there is roughly a 50% chance that the user will install App B before App A.

Here is where things go off the rails.

The user is not informed about App B's request for the custom permission.

The user will be informed about any platform permissions that the app requests via other `<uses-permission>` elements. If there are none, the user is told that App B requests no permissions... despite the fact that it does.

When the user installs App A, the same thing occurs. Of course, since App A does not have a `<uses-permission>` element, this is not all that surprising.

However, at this point, **even though the user was not informed**, App B holds the custom permission and can access the secured component.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This is bad enough when both parties are ethical. App B could be a piece of malware, though, designed to copy the data from App A, ideally without the user's knowledge. And, if App B is installed before App A, that would happen.

So, going to the two questions:

1. The user is **not** informed about App B's request for the permission...
2. ...but App B gets it anyway and can access the secured component

The Downgraded-Level Malware Case (B, Then A, Again)

You might think that the preceding problem would only be for `normal` or `dangerous` protection levels. If App A defines a permission as requiring a matching `signature`, and App A marks a component as being defended by that permission, Android must require the signature match, right?

Wrong.

The behavior is identical to the preceding case. Android does not use the *defender's* protection level. It uses the *definer's* protection level, meaning the protection level of whoever was installed first and had the `<permission>` element.

So, if App A has the custom permission defined as `signature`, and App B has the custom permission defined as `normal`, if App B is installed first, the behavior is as shown in the preceding section:

1. The user is **not** informed about App B's request for the permission...
2. ...but App B gets it anyway and can access the secured component, despite the signatures not matching

The Peer Apps Case With a Side Order of C

What happens if we add App C back into the mix? Specifically, what if App B is installed first, then App A, then App C?

When App C eventually gets installed, the user is prompted for the custom permission that App C requests via `<uses-permission>`. However, the description that the user sees is from App B, the one that first defined the custom `<permission>`. Moreover, the protection level is whatever App B defined it to be. So if App B downgraded the protection level from App A's intended `signature` to be `normal`,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

App C can hold that permission and access the secured App A component, even if it is signed by another signing key.

Not surprisingly, the same results occur if you install App B, then App C, then App A.

Behavior Analysis

The behavior exhibited in these scenarios is consistent with two presumed implementation "features" of Android's permission system:

1. First one in wins. In other words, the first app (or framework, in the case of the OS's platform permissions) that defines a `<permission>` for a given `android:name` gets to determine what the description is and what the protection level is.
2. The user is only prompted to confirm a permission if the app being installed has a `<uses-permission>` element, the permission was already defined by some other app, and the protection level is not `signature`.

Risk Assessment

The "first one in wins" rule is a blessing and a curse. It is a curse, insofar as it opens up the possibility for malware to hold a custom permission without the user's awareness of that, and even to downgrade a `signature`-level permission to `normal`. However, it is a blessing, in that the malware would have to be installed first; if it is installed second, either its request to hold the permission will be seen by the user (`normal` or `dangerous`) or the request to hold the permission will be rejected (`signature`).

This makes it somewhat unlikely for a piece of malware to try to sneakily make off with data. Eventually, if enough users start to ask publicly why App B needs access to App A's data (for cases where App A was installed first and the user knows about the permission request), somebody in authority may eventually realize that this is a malware attack. Of course, "eventually" may be a rather long time.

However, there are some situations where Android's custom permission behavior presents risk even greater than that. If the attacker has a means of being *sure* that their app was installed first, they can hold any permission from any third-party app they want to that was known at install time.

For example:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Somebody could sell a used Android device, and the buyer could neglect to factory-reset it, and the malware could be installed by the seller
- Somebody could sell a used Android device with a ROM mod preinstalled, based off of a normal ROM mod (e.g., CyanogenMod), but with an additional bit of malware installed, to prevent a factory reset from foiling the attack'
- Somebody could distribute devices to users who might think the device is "factory clean" and not laden with malware (e.g., devices given as gifts)
- Somebody could distribute devices to users who might think that the pre-installed malware is actually a legitimate app (e.g., devices given to employees by an employer wishing to monitor usage by examining protected data from third-party apps)

Android 5.0's "Fix"

Android 5.0 now prevents two apps from defining the same `<permission>` ("same" based on `android:name`) unless they are signed by the same signing key. First one in wins; the second app installation will fail.

On the plus side:

- This solves the security problem, as an attacker (B) cannot get at a defender's (A's) data by virtue of having been installed first, as A simply cannot be installed in this case.
- This has no impact on developers using `signature`-level `<permission>` elements for their own app suite.

However, it does pose significant limitations on legitimate public uses of custom `<permission>` elements. Only the defender should have the `<permission>` element now. Some client of the defender's app (C) should not have the `<permission>` element and should simply rely upon the fact that the defender should be installed first. If the client were to define the `<permission>`, then either the client *or* the defender cannot be installed, which is pointless.

This has usability issues:

- A client should check, on first run of their app, if an expected defender (and its `<permission>` element) exists. If not, the client should alert the user to this fact and perhaps stop the app from proceeding further. The user would have to uninstall the client, install the defender, then reinstall the client, to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

get everything working properly, and the more the user uses the client app, the more painful the uninstall might be.
- It is impossible for two apps to be clients of each other. By definition, one app has to be installed first and the other second, which means only the first-to-be-installed app can have a custom `<permission>`. If Facebook wanted to hold a custom Twitter permission, and Twitter wanted to hold a custom Facebook permission, one of them is out of luck — if Facebook is installed first, it cannot request Twitter's permission (as it does not yet exist) nor can it define Twitter's permission (as if it does, Twitter cannot be installed). This *might* be able to be overcome for apps that are pre-loaded as part of a ROM mod or other custom Android build.

And, of course, this fix is only for Android 5.0 and above.

Mitigation Using PermissionUtils

The "first one in wins" rule also leads us to a mitigation strategy: On first run of our app, see if any other app has defined permissions that we have defined. If that has happened, then we are at risk, and take appropriate steps. If, however, no other app has defined our custom permissions, then the Android permission system should work for us, and we can proceed as normal.

The CWAC-Security library provides some helper code, in the form of the `PermissionUtils` class, to detect other apps defining the same custom permissions that you define.

The idea is that you call `checkCustomPermissions()` — a static method on `PermissionUtils` — on the first run of your app. It will return details about what other apps have already defined custom permissions that your app defines. If `checkCustomPermissions()` returns nothing, you know that everything is fine, and you can move ahead. Otherwise, you can:

- Check to see if the offending app is on some whitelist, or otherwise meets criteria that suggests that it is OK
- Alert the user, indicating that these already-installed apps will have access to your app secured components
- Upload details about the offending apps to your server, so you can try to track down whether they are legitimate users of some API that you are exposing or are malware
- Whatever else you feel is necessary

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Example: Permission Proxy

The section on `ContentProvider` proxy plugins involves the use of a custom `signature`-level permission, to secure communications between the proxy and the host app that uses the proxy.

The idea is that the proxy holds some permission (e.g., `READ_CONTACTS`) and proxies data to some `ContentProvider` protected by that proxy (e.g., `CallLog`). The host app, rather than holding the permission and accessing the protected `ContentProvider` directly, can talk to the proxy. That way, the user only needs to grant permission if they elect to install the proxy; otherwise, the host app is blocked from having access to the protected content.

However, to prevent arbitrary other apps from using the proxy themselves, the host and proxy agree on a custom `signature`-level permission. The proxy defends itself using that permission, and the host requests the permission. In theory, this would limit communications with the proxy to only be from the host, or from other apps signed with the same signing key as the proxy and host use.

But, as is described above, another app could define the *same* permission, with a `normal` protection level. If that other app is installed first, not only can any other app access the proxy just by requesting the permission, but the attacker could have requested the same permission that it defined, so the user is unaware that the attacker holds this permission.

Hence, these proxies need to use some defensive measures, and the samples shown in this book employ `PermissionUtils` from the CWAC-Security library to do just that.

What the Proxy Does

The proxy is a `ContentProvider`. Specifically, there is an `AbstractCPProxy` subclass of `ContentProvider` that does the "heavy lifting", and a `CallLogProxy` subclass of `AbstractCPProxy` that handles some of the details of proxying the `CallLog` versus something else.

In `onCreate()` of the `AbstractCPProxy`, we use `PermissionUtils` and `checkCustomPermissions()` to determine whether or not anything was installed before us, that defined our custom permission, *other* than our known host app:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public boolean onCreate() {
    SharedPreferences prefs=
        PreferenceManager.getDefaultSharedPreferences(getContext());

    if (prefs.getBoolean(PREFS_FIRST_RUN, true)) {
      SharedPreferences.Editor editor=
          prefs.edit().putBoolean(PREFS_FIRST_RUN, false);

      HashMap<PackageInfo, ArrayList<PermissionLint>> entries=
          PermissionUtils.checkCustomPermissions(getContext());

      for (Map.Entry<PackageInfo, ArrayList<PermissionLint>> entry :
entries.entrySet()) {
        if
(!"com.commonsware.android.cpproxy.consumer".equals(entry.getKey().packageName))
{
          tainted=true;
          break;
        }
      }

      editor.putBoolean(PREFS_TAINTED, tainted).apply();
    }
    else {
      tainted=prefs.getBoolean(PREFS_TAINTED, true);
    }

    return(true);
  }
```

We use `SharedPreferences` to hold onto two key pieces of data:

1. Have we already done the check, as determined by `PREFS_FIRST_RUN`? If yes, we can just look up the results of the previous check. This is not merely an optimization — we do not have to worry about apps installed *after* us somehow redefining our custom permission.
2. When we did the check, did we find some package that had been installed before us, other than the host, that defined our custom permission, as determined by `PREFS_TAINTED`?

The actual check is accomplished by calling `checkCustomPermissions()` and iterating over the results. If there is an entry in the `HashMap` that represents a package other than ours, our environment is tainted.

The implementation in the book then uses the `tainted` data member in a `checkTainted()` private method:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
private void checkTainted() {
  if (tainted) {
    throw new RuntimeException(getContext().getString(R.string.tainted_abort));
  }
}
```

This is called at the top of each `ContentProvider` method that we are proxying, such as `insert()`:

```
@Override
public Uri insert(Uri uri, ContentValues values) {
  checkTainted();

  return(getContext().getContentResolver().insert(convertUri(uri),
                                                  values));
}
```

The result is that if we feel that our environment is compromised, we fail any attempt to use the proxy.

Note that the proxy makes no attempt to confirm that the host app really *is* the host app, versus some other app with the same package name, perhaps distributed through other channels. We could augment the proxy with additional logic to handle that case, covered elsewhere in this book, if we wanted.

Also, we could, in theory, use `Binder.getCallingUid()` to confirm whether the request did come from the host app, and in that case, allow the proxy to do its work, failing in all other cases. We could even consider jettisoning the custom permission in this case, as if we know the UID of the other party, we can validate it instead of relying on a permission as a means of validation. However, that only works well in cases where the list of possible valid callers is knowable inside the app — this is fine for host-and-plugin or similar sorts of "hub-and-spoke" architectures but may be impractical in other cases.

What the Provider Could Do

The proxy has no decent means of alerting the user as to the reason for the lockdown. After all, it is a `ContentProvider`, not an `Activity`. In principle, it could use a `Notification`.

Another approach is to have the host app perform the same sorts of checks as does the proxy, and use that information to inform the user on first run of the app.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Restricted Profiles and UserManager

Android 4.2 introduced the concept of having multiple distinct users of a tablet. Each user would get their own portion of internal and external storage, as if they each had their own tablet.

Android 4.3 extends this a bit further, with the notion of setting up restricted profiles. As the name suggests, a restricted profile is restricted, in terms of what it can do on the device. Some restrictions will be device-wide (e.g., can the user install apps?), and some restrictions will be per-app. You can elect to allow your app to be restricted, where you define the possible ways in which your app can be restricted, and the one setting up the restricted profile can then configure the desired options for some specific profile.

This chapter will explain how users set up these restricted profiles, what you can learn about the device-wide restrictions, and how you can offer your own restrictions for your own app.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the chapter on files and its section on multiple user accounts.

Android Tablets and Multiple User Accounts

The theory is that tablets are likely to be shared, whether among family members, among team members in a business, or similar sorts of group settings. There are three levels of "user" in an Android 4.3+ tablet that we will need to consider.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Primary User

The primary user is whoever first set up the tablet after initial purchase. In a family, this is probably a parent; in a corporate setting, this might be an IT administrator.

Prior to Android 4.2, there was only one user per device, and that user could (generally) do anything. In Android 4.2+, the primary user holds this role.

One thing that the primary user can do is set up other users, via the Users option in the Settings app:

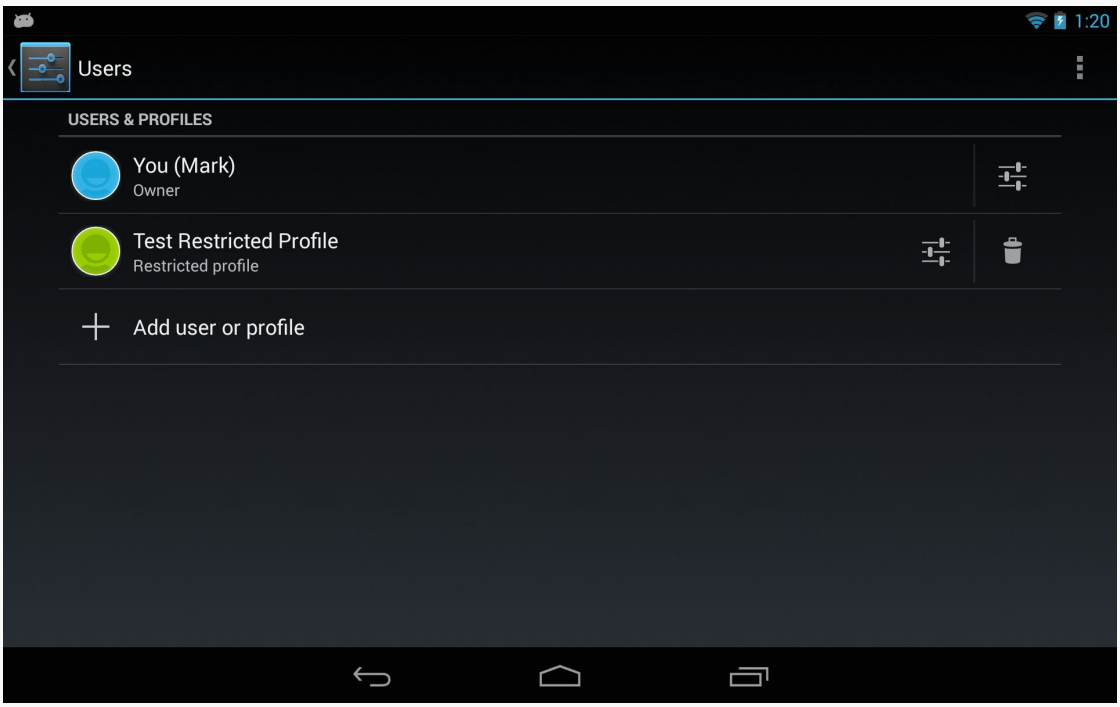


Figure 652: Users Screen in Settings

Tapping the “Add user or profile” entry allows the primary user to set up another user or restricted profile:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

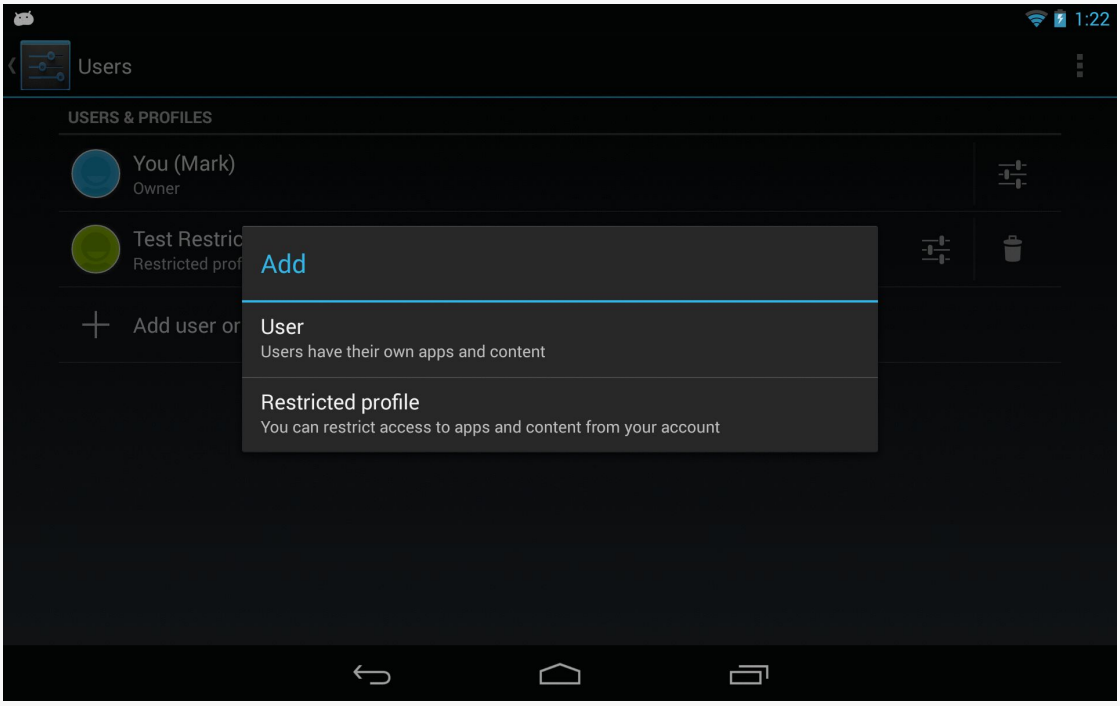


Figure 653: Add Dialog in Users Screen in Settings

Secondary User

Choosing “User” from the Add dialog will define a secondary user of the device. This user has much of the same control as the primary user, in terms of being able to install and run whatever apps are desired.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

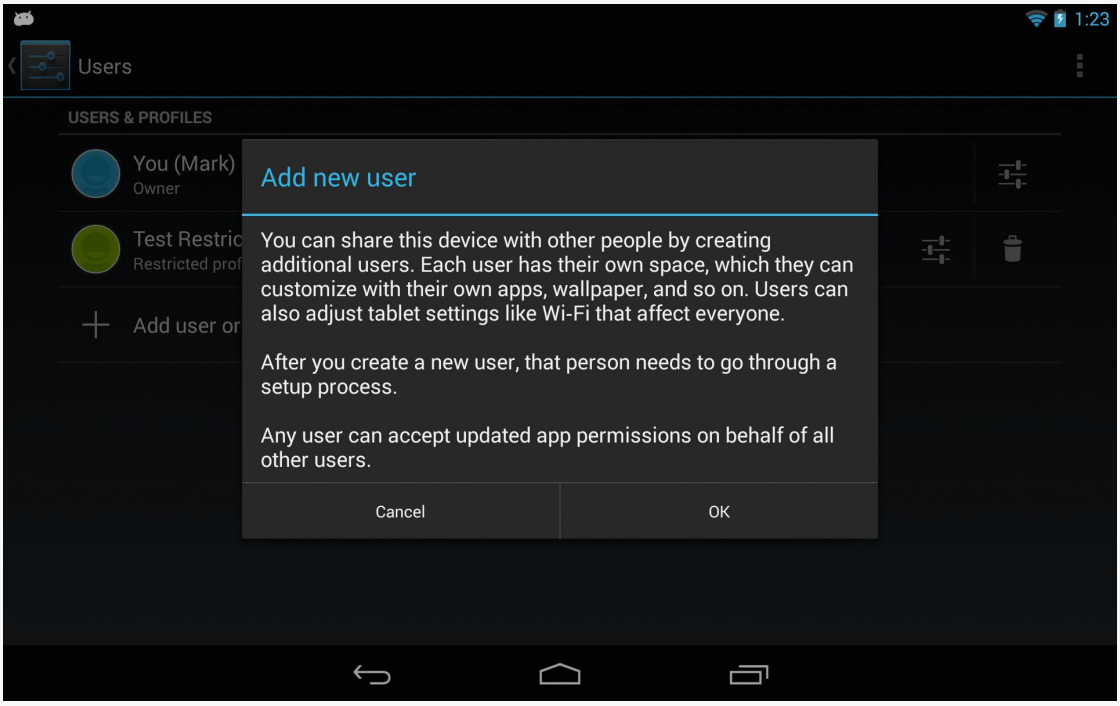


Figure 654: Add New User Warning Dialog in Users Screen in Settings

Restricted Profile

A restricted profile is akin to a secondary user, in that it gets its own separate portion of internal and external storage. Beyond that, though, the primary user can further configure what the restricted profile can access:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

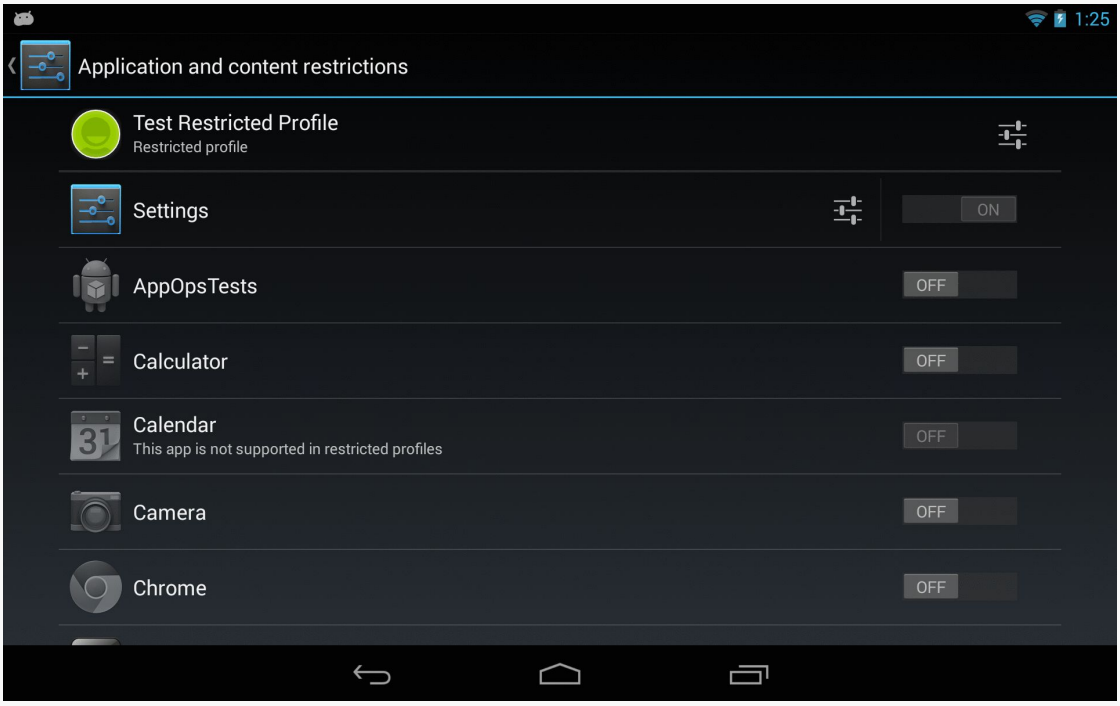


Figure 655: Restricted Profile Configuration Screen in Settings

The bulk of the restricted profile configuration screen is a list of apps, with `Switch` widgets to allow the primary user to allow or deny access to each app.

Some apps will have the "settings" icon to the left of the `Switch`. Tapping that will either bring up a dedicated activity for restricting operations within that app, or it will add new rows to the list with individual restriction options for that app. For example, tapping the settings icon for the Settings app adds a row where the primary user can block location sharing:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

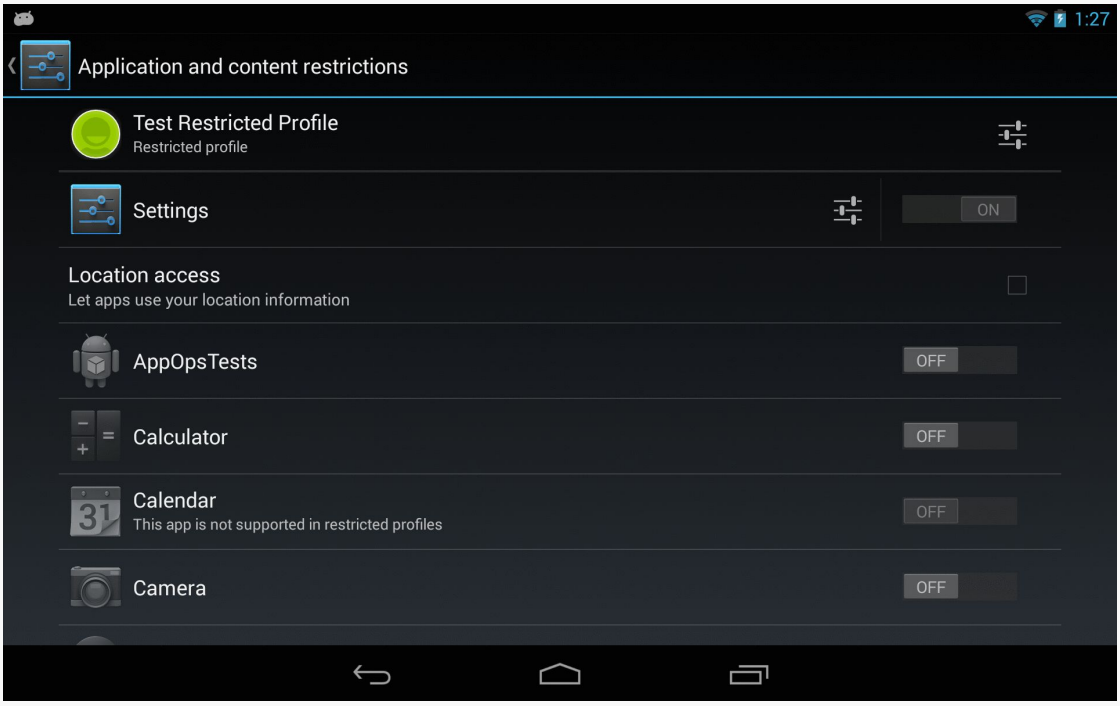


Figure 656: Location Sharing Restrictions

The “settings” icon in the first row, for the profile itself, will allow the primary user to control things for the entire profile, notably its name.

Switching to the restricted profile (e.g., via the lockscreen) will show the constrained set of available apps:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

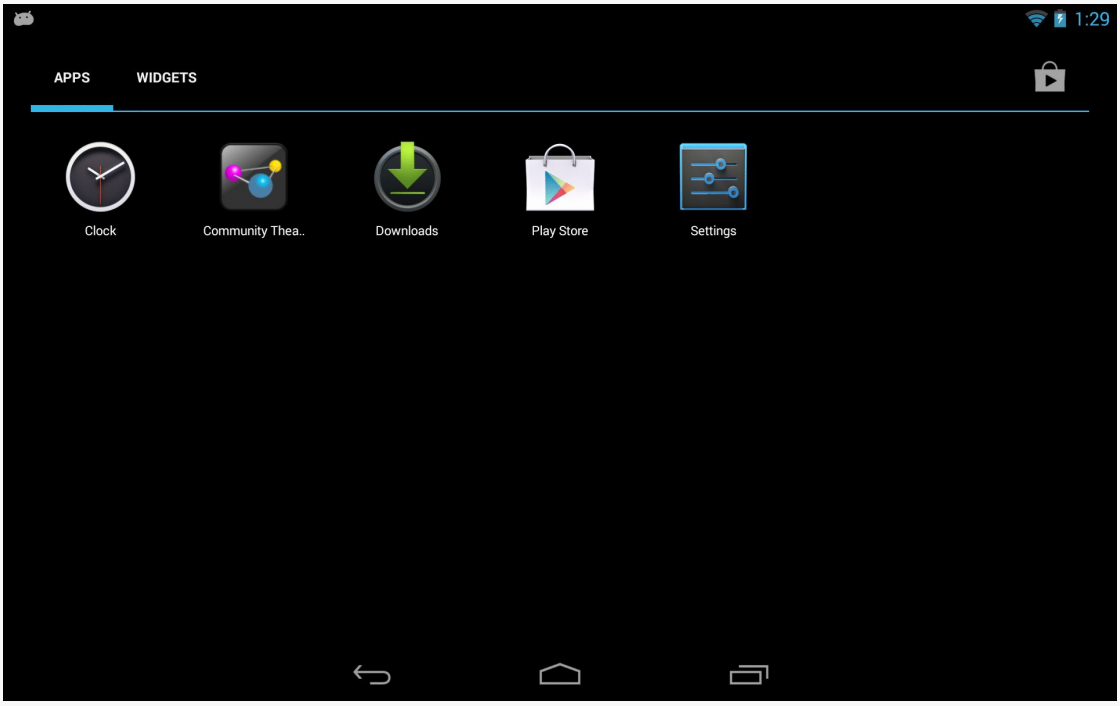


Figure 657: Apps in a Restricted Profile

Determining What the User Can Do

Your app can find out what device-level restrictions were placed on the current user by means of the `UserManager` system service. Specifically, as you can see in `MainActivity` of the [RestrictedProfiles/Device] sample project, all you need to do is:

- Acquire an instance of a `UserManager` by calling `getSystemService()` on a `Context`, passing in `USER_SERVICE` as the service's name
- Calling `getUserRestrictions()` on the `UserManager`:

```
package com.commonsware.android.profile.device;

import android.app.Activity;
import android.os.Bundle;
import android.os.UserManager;
import android.widget.Toast;

public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    UserManager mgr=(UserManager)getSystemService(USER_SERVICE);
    Bundle restrictions=mgr.getUserRestrictions();

    if (restrictions.keySet().size() > 0) {
      setContentView(R.layout.activity_main);

      RestrictionsFragment f=

(RestrictionsFragment)getFragmentManager().findFragmentById(R.id.contents);

      f.showRestrictions(restrictions);
    }
    else {
      Toast.makeText(this, R.string.no_restrictions, Toast.LENGTH_LONG)
           .show();
      finish();
    }
  }
}
```

`getUserRestrictions()` returns a `Bundle`, whose keys are documented on `UserManager` for various device-level restrictions that theoretically can be placed on the user. Here, "theoretically" means that while `UserManager` documents several `DISALLOW_*` constants, only two seem to be directly accessible to the primary user for configuration via Settings:

- `DISALLOW_MODIFY_ACCOUNTS`, to prevent a restricted profile from, among other things, modifying restricted profiles
- `DISALLOW_SHARE_LOCATION`, to prevent the apps run in this restricted profile from gathering location data

`MainActivity` examines the `Bundle` and, if it is empty, just displays a `Toast` and exits via `finish()`. This is the behavior you will see if you run this sample app on a non-restricted profile, such as the primary user. If, however, the `Bundle` has one or more keys, we inflate an `activity_main` layout that contains a `RestrictionsFragment` in a `<fragment>` element:

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/contents"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  class="com.commonsware.android.profile.device.RestrictionsFragment"/>
```

We then retrieve the `RestrictionsFragment` from the `FragmentManager` and call `showRestrictions()` on it, passing in the `Bundle`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`RestrictionsFragment` is a `ListFragment` employing a custom `RestrictionsAdapter`. The `RestrictionsAdapter` wraps around the `Bundle` and an `ArrayList` of its keys. The `RestrictionsAdapter` constructor creates the `ArrayList` by sorting the `keySet()` of the `Bundle`. `getView()` on `RestrictionsAdapter` lets the superclass handle inflating the row (`android.R.layout.simple_list_item_1`), then puts an icon on the right side by using `setCompoundDrawablesWithIntrinsicBounds()`, which can tuck a drawable resource onto any of the four sides of a `TextView`.

The resulting list will show green icons for keys where the `Bundle` has stored a `true` `Boolean` value, and a red icon for `false`:

Figure 658: Default Device Restrictions, on a Nexus 7 (2013)

Since the keys are negative in tone (e.g., `DISALLOW_MODIFY_ACCOUNTS`), `true` means that the restriction is enforced and the underlying operation (e.g., modifying accounts) *cannot* be done.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Impacts of Device-Level Restrictions

Your app's functionality may be limited by these device-level restrictions. This section outlines some of the results you should expect from a restricted profile.

Restricting Location Access

If a restricted profile is prevented from sharing the device's location with apps, those apps simply will not receive location updates. There is no good way to detect this via the location API (e.g., `isProviderEnabled()` returns `true`), so you will have to detect this via `getUserRestrictions()` on `UserManager` as noted above.

Uninstalling Apps

Even without specific configuration, the restricted profile can only uninstall apps that the are available to that profile. However, since apps are really shared between profiles, this only removes that app from the restricted profile; it does not actually uninstall the app from the device as a whole.

Enabling Custom Restrictions

As noted earlier, the list of apps that is shown on the restricted profile configuration screen in Settings can have "settings" icons. The Settings app itself will have a settings icon, to allow the primary user to configure device-level restrictions.

But, what if you want *your* app to have such a settings icon? Maybe it makes sense for your app to allow the primary user to restrain restricted profiles from doing certain things within your app:

- Block in-app purchases
- Only show certain categories of content, not the full roster
- Only allow operation during certain times of the day

The means by which the Settings app restricts profiles is also available to you. You can declare to Android what aspects of your app can be restricted. Android will then collect that restriction data for you. Your app, at runtime, can then determine what restrictions are in place (if any) and take appropriate steps.

All of this will be illustrated using the `RestrictedProfiles/App` sample project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Stating Your Restrictions

The biggest thing that you need to do to restrict your app is teach Android how to collect restrictions. In other words, you need to tell Android what to do when the user taps that settings icon in the restricted profile entry for your app.

You have two major options:

- Provide a list of the restrictions that Android should render and collect itself, or
- Provide an `Intent` that can be used to start up an activity of your own design where you collect those restrictions

Either approach will require you to set up a manifest-registered `BroadcastReceiver`, set to respond to the `android.intent.action.GET_RESTRICTION_ENTRIES` action:

```
    <receiver android:name="RestrictionEntriesReceiver">
      <intent-filter>
        <action android:name="android.intent.action.GET_RESTRICTION_ENTRIES"/>
      </intent-filter>
    </receiver>
```

That `BroadcastReceiver` will be called with `sendOrderedBroadcast()`, not so much to affect ordering, but to allow the `BroadcastReceiver` to send back a result via its `setResultExtras()` method. This provides a `Bundle` that the broadcaster can eventually retrieve, in this case providing details of what restrictions we wish to collect from the primary user to restrict the profile.

Option #1: RestrictionEntry List

To collect restrictions the way the Settings app does — with restriction rows appearing below your app in the restricted profile screen in Settings – your `BroadcastReceiver` will need to put an entry into the return `Bundle`, under the key of `EXTRA_RESTRICTIONS_LIST` (a constant defined on the `Intent` class). The value needs to be an `ArrayList` of `RestrictionEntry` objects, with each `RestrictionEntry` describing one restriction to collect.

Another thing that the `RestrictionEntry` objects contain is their current value. Android itself retains these values and supplies them to your `BroadcastReceiver` via an `EXTRA_RESTRICTIONS_BUNDLE` extra on the incoming `Intent`. Your app needs to use those current values when constructing its list of `RestrictionEntry` objects to return.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

So, let's take a look at RestrictionEntriesReceiver, the receiver we have set up to handle the android.intent.action.GET_RESTRICTION_ENTRIES action for this sample app.

The entry point for RestrictionEntriesReceiver is onReceive(), as it is for any basic BroadcastReceiver:

```
@Override
public void onReceive(Context ctxt, Intent intent) {
  Bundle current=
      (Bundle)intent.getParcelableExtra(Intent.EXTRA_RESTRICTIONS_BUNDLE);
  ArrayList<RestrictionEntry> restrictions=
      new ArrayList<RestrictionEntry>();

  restrictions.add(buildBooleanRestriction(ctxt, current));
  restrictions.add(buildChoiceRestriction(ctxt, current));
  restrictions.add(buildMultiSelectRestriction(ctxt, current));

  Bundle result=new Bundle();

  result.putParcelableArrayList(Intent.EXTRA_RESTRICTIONS_LIST,
                                restrictions);

  setResultExtras(result);
}
```

In onReceive(), RestrictionEntriesReceiver pulls out the Bundle of current restrictions, by retrieving the EXTRA_RESTRICTIONS_BUNDLE extra from the Intent passed into onReceive(). Note that this Bundle could very well be empty, if this is the first time we are being asked for restrictions.

RestrictionEntriesReceiver creates an empty ArrayList of RestrictionEntry objects, then calls a series of builder methods to create a total of three such RestrictionEntry objects, adding each to the list. onReceive() goes on to create a Bundle representing the results to be returned, packages the ArrayList in that Bundle under the EXTRA_RESTRICTIONS_LIST key, and returns that Bundle to the caller by means of setResultExtras().

The three builder methods are each responsible for defining a single RestrictionEntry, including populating it with the current value from the current Bundle.

There are three types of RestrictionEntry, for boolean, single-selection lists ("choice"), and multi-selection lists. The RestrictionEntry constructor takes two parameters:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- The `String` key under which we will later retrieve this restriction value
- The current value of the restriction

The current value is:

- A `boolean` for boolean restrictions
- A `String` for choice restrictions
- A `String` array for multi-select restrictions

Our first builder, `buildBooleanRestriction()`, populates and returns a `RestrictionEntry` designed to collect a boolean value from the primary user, via a `CheckBox`:

```
  private RestrictionEntry buildBooleanRestriction(Context ctxt,
                                                   Bundle current) {
    RestrictionEntry entry=
        new RestrictionEntry(RESTRICTION_BOOLEAN,
                             current.getBoolean(RESTRICTION_BOOLEAN,
                                                false));

    entry.setTitle(ctxt.getString(R.string.boolean_restriction_title));
    entry.setDescription(ctxt.getString(R.string.boolean_restriction_desc));

    return(entry);
  }
```

`buildBooleanRestriction()` retrieves the current value from `current` `Bundle` to use with the `RestrictionEntry` constructor. In this case, if there is no such entry in the `Bundle`, the overall default value is `false`.

Each `RestrictionEntry` can have a title (`setTitle()`), supplying a string which will be displayed to describe what this restriction is. A boolean restriction can also have a description (`setDescription()`), containing another string with a bit more text. Note that, at the present time, the other two types of restrictions will ignore any description that you include. Also note that the values supplied to `setTitle()` and `setDescription()` need to be strings, and so if you wish to use a string resource, you will need to get the actual string value yourself via `getString()`.

The remaining two builder methods have a similar structure:

```
  private RestrictionEntry buildChoiceRestriction(Context ctxt,
                                                  Bundle current) {
    RestrictionEntry entry=
        new RestrictionEntry(RESTRICTION_CHOICE,
                             current.getString(RESTRICTION_CHOICE));
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    entry.setTitle(ctxt.getString(R.string.choice_restriction_title));
    entry.setChoiceEntries(ctxt, R.array.display_values);
    entry.setChoiceValues(ctxt, R.array.restriction_values);

    return(entry);
  }

  private RestrictionEntry buildMultiSelectRestriction(Context ctxt,
                                                       Bundle current) {
    RestrictionEntry entry=
        new RestrictionEntry(RESTRICTION_MULTI,
                             current.getStringArray(RESTRICTION_MULTI));

    entry.setTitle("A Multi-Select Restriction");
    entry.setChoiceEntries(ctxt, R.array.display_values);
    entry.setChoiceValues(ctxt, R.array.restriction_values);

    return(entry);
  }
```

As with a `ListPreference`, you provide two string arrays to the `RestrictionEntry`, representing the values the primary user sees (`setChoiceEntries()`) and the corresponding values to be supplied to your app based upon the choice(s) (`setChoiceValues()`). You can supply these either as Java string arrays or as `<string-array>` resources – `RestrictionEntriesReceiver` goes with the latter approach.

Option #2: Custom Restriction Activity

It may be that what you want to collect, in terms of restrictions, cannot readily be represented in the form of `Switch` widgets and list dialogs. For example, to restrict use of your app based on time, it would be nice to use a `TimePickerDialog` or the equivalent.

The alternative to returning an `EXTRA_RESTRICTIONS_LIST` roster of `RestrictionEntry` objects from your `BroadcastReceiver` is to have the result `Bundle` contain `EXTRA_RESTRICTIONS_INTENT`. This key should point to an `Intent` that identifies the activity that you want to start up when the user taps the settings icon. Android will call `startActivityForResult()` on that `Intent` when the user taps on the settings icon.

Your job is to collect the restrictions from the user, using the `EXTRA_RESTRICTIONS_BUNDLE` from the incoming `Intent` to pre-populate your activity, if desired. When the user is done, you should call `setResult()`, passing in an `Intent` that contains another `EXTRA_RESTRICTIONS_BUNDLE` with the revised data,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

or optionally a `EXTRA_RESTRICTIONS_LIST` (with the `RestrictionEntry` objects containing the values to be used).

What the Primary User Sees

Given the `RestrictionEntriesReceiver` described above, when the primary user goes to configure a restriction profile, your app will appear with a settings icon next to it:

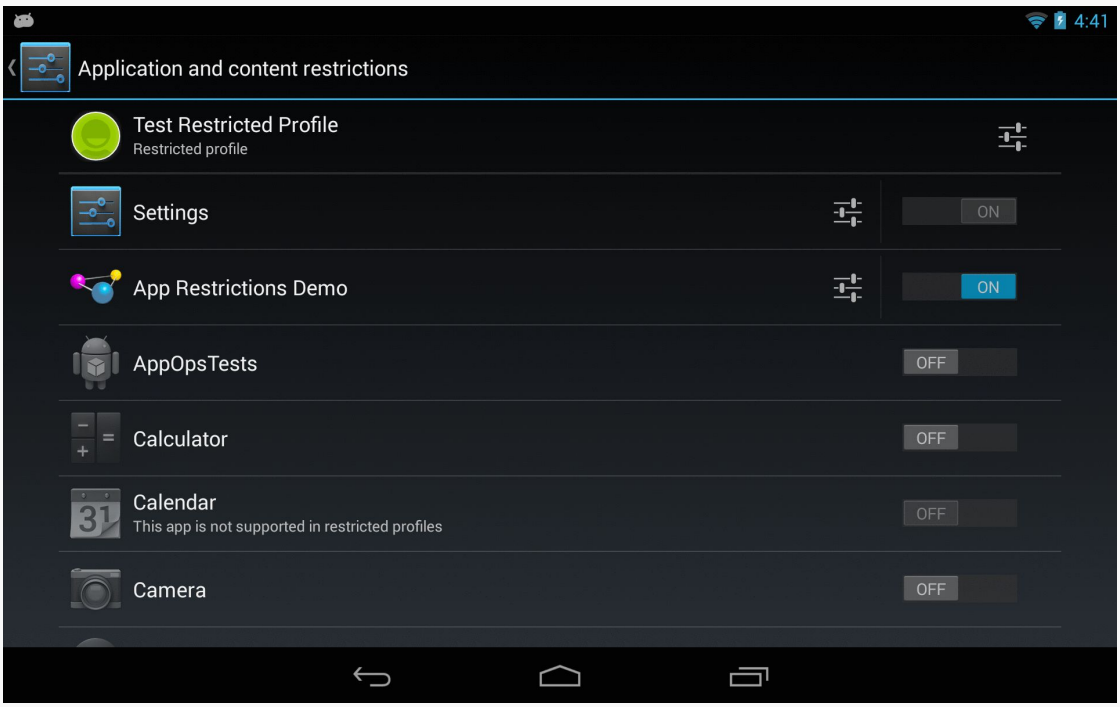


Figure 659: Restricted Profile, Showing App Settings Icon

Tapping that settings icon will "unfold" and display the restrictions that you configured via the `RestrictionEntry` objects:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

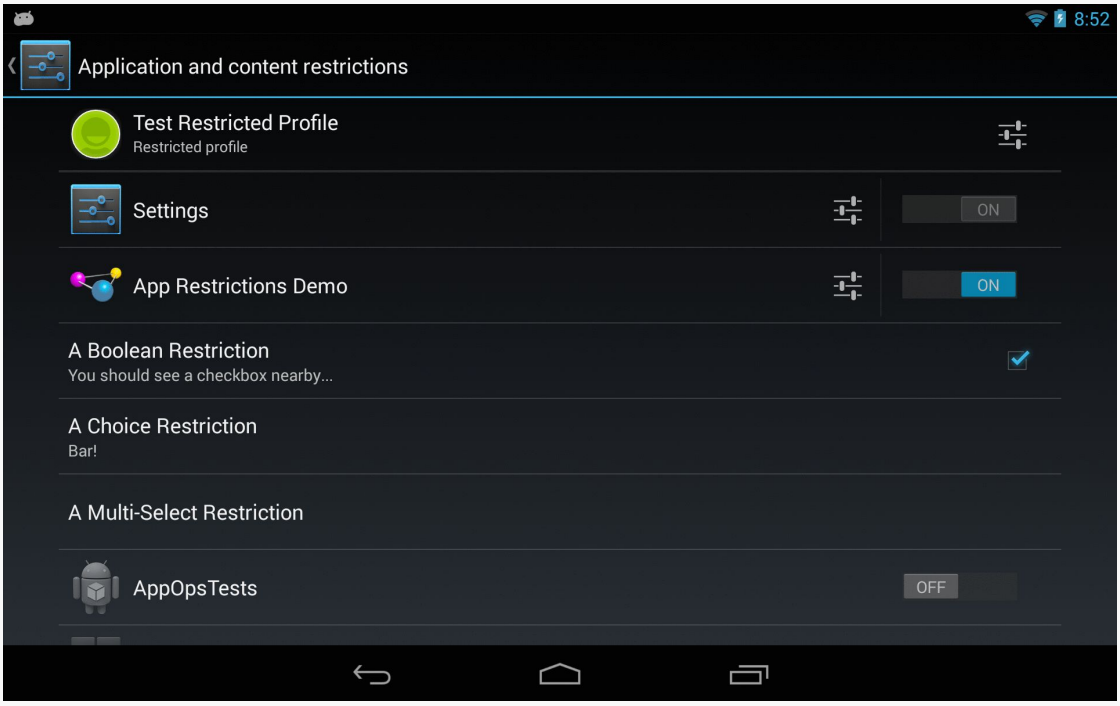


Figure 660: Restricted Profile, Showing App Restrictions

The primary user can then interact with your restrictions, toggling checkboxes and popping up the list dialogs:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

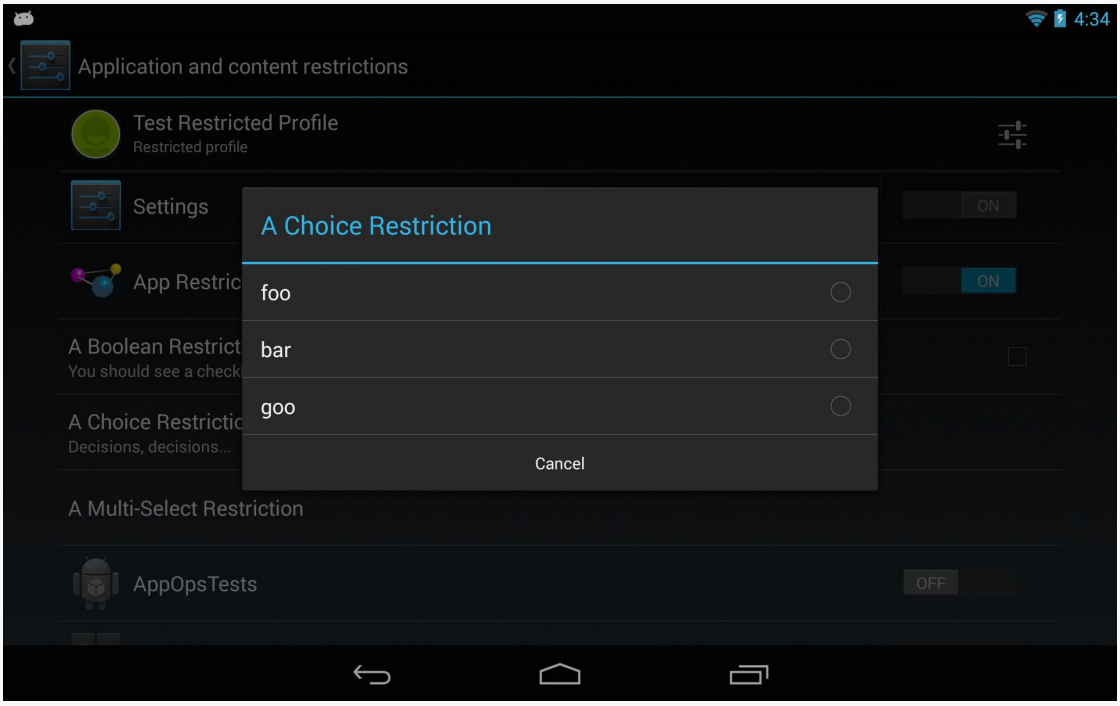


Figure 661: Restricted Profile, Showing Choice Restriction

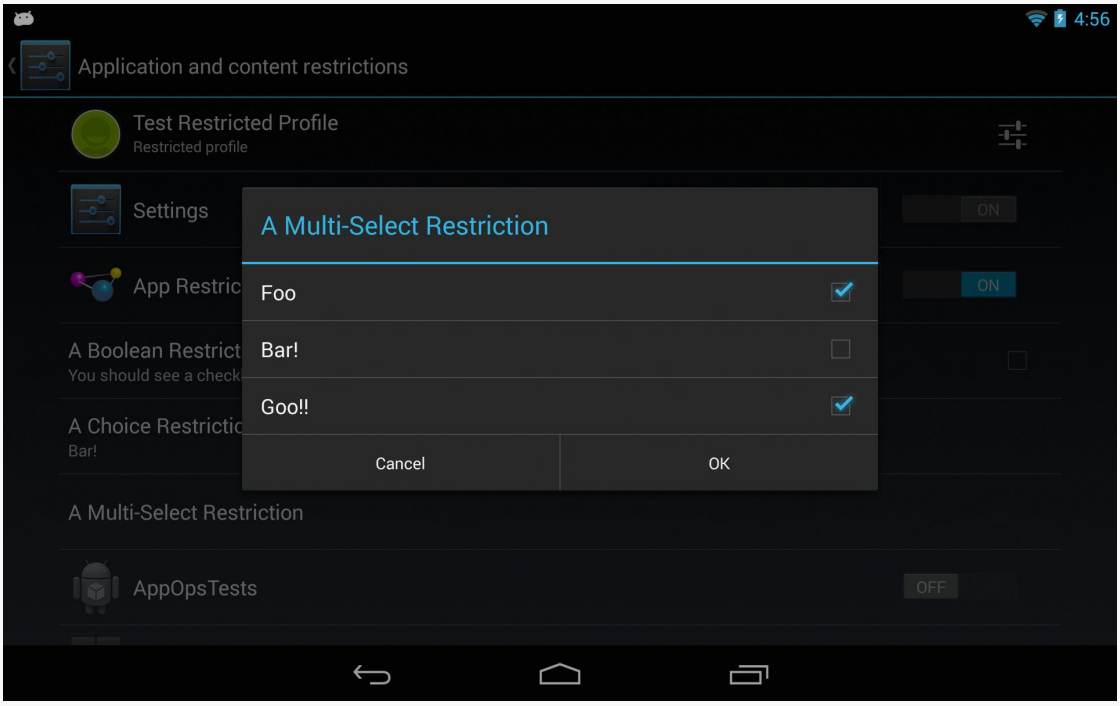


Figure 662: Restricted Profile, Showing Multi-Select Restriction

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Finding Out the Current Restrictions

Now, the rest of your app needs to find out what restrictions are placed upon it, so behavior can be tailored accordingly. To do this, call `getApplicationRestrictions()` on `UserManager`, passing in your package name, as seen here in `MainActivity`:

```
package com.commonsware.android.profile.app;

import android.app.Activity;
import android.os.Bundle;
import android.os.UserManager;
import android.widget.Toast;

public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    UserManager mgr=(UserManager)getSystemService(USER_SERVICE);
    Bundle restrictions=
        mgr.getApplicationRestrictions(getPackageName());

    if (restrictions.keySet().size() > 0) {
      setContentView(R.layout.activity_main);

      RestrictionsFragment f=

(RestrictionsFragment)getFragmentManager().findFragmentById(R.id.contents);

      f.showRestrictions(restrictions);
    }
    else {
      Toast.makeText(this, R.string.no_restrictions, Toast.LENGTH_LONG)
           .show();
      finish();
    }
  }
}
```

This `Bundle` could be empty, or it could have values specified by the primary user to restrict the profile that is running your app.

In the case of this sample, we once again set up a `RestrictionsAdapter` to show the results, if the `Bundle` is not empty. However, our adapter is a bit more complicated, as there are more than boolean restrictions now. `getView()` has been updated to handle all three possible restrictions, showing the icon for the boolean restriction, and showing the value(s) from the lists in the other restrictions:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.profile.app;

import android.app.ListFragment;
import android.os.Bundle;
import android.text.TextUtils;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.TextView;
import java.util.ArrayList;
import java.util.Collections;

public class RestrictionsFragment extends ListFragment {
  public void showRestrictions(Bundle restrictions) {
    setListAdapter(new RestrictionsAdapter(restrictions));
  }

  class RestrictionsAdapter extends ArrayAdapter<String> {
    Bundle restrictions;

    RestrictionsAdapter(Bundle restrictions) {
      super(getActivity(), android.R.layout.simple_list_item_1,
            new ArrayList<String>());

      ArrayList<String> keys=
          new ArrayList<String>(restrictions.keySet());

      Collections.sort(keys);
      addAll(keys);

      this.restrictions=restrictions;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      TextView row=
          (TextView)super.getView(position, convertView, parent);
      String key=getItem(position);

      if (RestrictionEntriesReceiver.RESTRICTION_BOOLEAN.equals(key)) {
        int icon=
            restrictions.getBoolean(key) ? R.drawable.ic_true
                : R.drawable.ic_false;

        row.setCompoundDrawablesWithIntrinsicBounds(0, 0, icon, 0);
      }
      else if (RestrictionEntriesReceiver.RESTRICTION_CHOICE.equals(key)) {
        row.setText(String.format("%s (%s)", key,
                                  restrictions.getString(key)));
      }
      else {
        String value=
            TextUtils.join(" | ", restrictions.getStringArray(key));
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        row.setText(String.format("%s (%s)", key, value));
      }

      return(row);
    }
  }
}
```

The result, when run on a restricted profile with restrictions placed upon our app, is to show those restrictions:

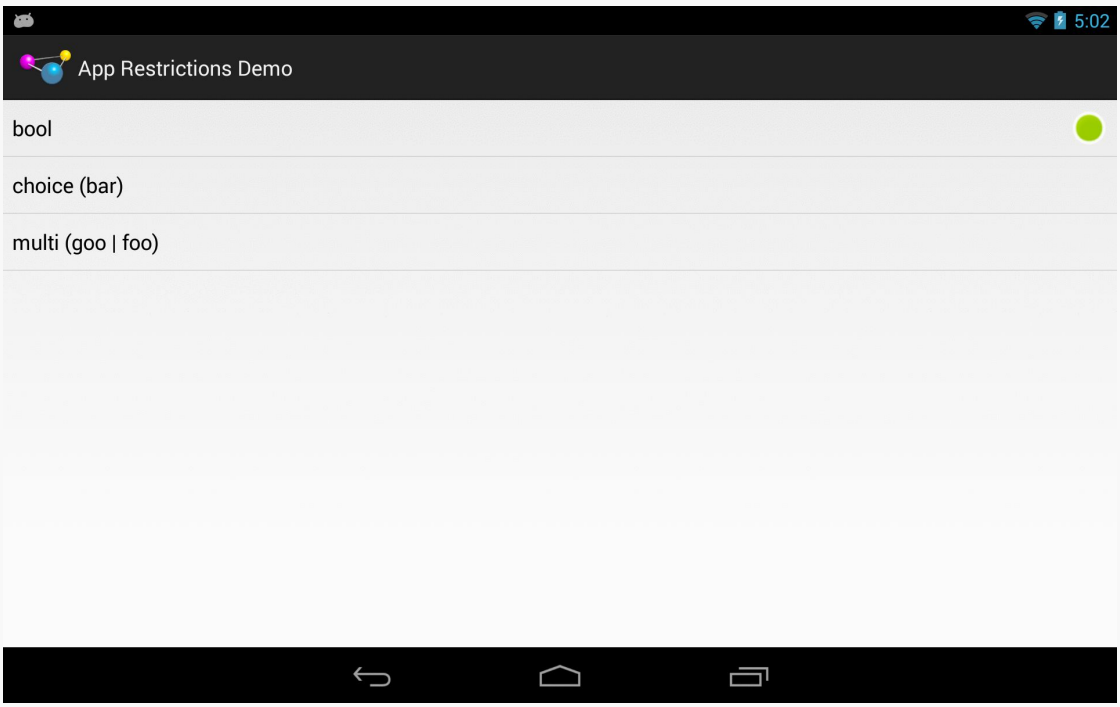


Figure 663: App Restrictions Demo, on a Restricted Profile

The Uninstall Bug

Note that there appears to be a bug in Android 4.3, where an uninstalled app's restrictions remain intact. Your app may continue to show up in the restricted profile, and if the user reinstalls the app later on, whatever original restrictions were placed on a restricted profile return.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Implicit Intents May Go "Boom"

The primary user of a tablet, when setting up a restricted profile, can control what apps are available to that profile. In many cases, if the user is setting up a restricted profile in the first place, the list of apps available to that profile will be fairly limited, such as only allowing a young child to access a few games and educational apps.

`startActivity()` always has the chance of throwing an `ActivityNotFoundException`. However, for certain `Intent` actions, we often ignore this possibility, because we are certain that there will be an app that can handle our request:

- All Android devices have Web browsers, right?
- All Android devices have some sort of mapping application, right?
- All Android devices let you pick a contact, right?

Now, with restricted profiles, you will need to deal with the `ActivityNotFoundException` case all of the time. You have three basic approaches for this:

1. Wrap all `startActivity()` and `startActivityForResult()` calls in a `try/catch` block that catches `ActivityNotFoundException` and intelligently handle the problem
2. Use `PackageManager` and `resolveActivity()` *before* trying to start the activity, where if `resolveActivity()` returns `null`, you know that there is no activity available to handle your desired operation
3. Switch out some of your `startActivity()` and `startActivityForResult()` calls for implementations in your app (e.g., embed Maps V2 rather than try to launch a potentially-nonexistent activity)

You might consider implementing a `safeStartActivity()` utility method that wraps up your particular plan, so you can debug it once.

The Future: App Ops?

Android 4.3 briefly had a leaked activity in the Settings app, that allows users to enable or disable operations on a per-app basis. These operations cover many common scenarios identified by permissions (e.g., read contacts, write contacts, access location), though it is not a pure one-to-one mapping. This activity is referred to as "App Ops", based upon the title shown in the action bar.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

While that activity has since been removed from the Settings app in AOSP, it is perhaps a sign of things to come. Also, Android 4.4 has introduced an `AppOpsManager` that presumably ties into this Settings activity, though that activity was not enabled in Android 4.4 either. One can imagine that primary users will be able to restrict specific operations for allowed apps for restricted profiles. So while App Ops is not an immediate concern for most Android developers, it is something to keep an eye on.

The author of this book maintains an App Ops developer FAQ with more information on the subject.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Tapjacking

On the whole, Android's security is fairly good for defending an app from another app. Between using Linux users and filesystems for protecting an application's files from other apps, to the use of custom permissions to control access to public interfaces, an application would seem to be relatively protected.

However, there is one attack vector that existed until Android 4.0.3: tapjacking. This chapter outlines what tapjacking is and what you can do about it to protect your app's users, for as long as you are supporting devices older than 4.0.3.

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- broadcast Intents
- service theory

What is Tapjacking?

Tapjacking refers to another program intercepting and inspecting touch events that are delivered to your foreground activity (or related artifacts, such as the input method editor). At its worst, tapjackers could intercept passwords, PINs, and other private data.

The term "tapjacking" seems to have been coined by Lookout Mobile Security, in a blog post that originally demonstrated this issue.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You might be wondering how this is possible. There are a handful of approaches to implementing this. The Lookout blog post cited perhaps the least useful approach: making a transparent `Toast`. The `Tapjacking/Jackalope` sample application will illustrate a far more troublesome implementation.

World War Z (Axis)

You may recall that there are three axes to consider with Android user interfaces. The X and Y axes are the ones you typically think about, as they control the horizontal and vertical positioning of widgets in an activity. The Z axis — effectively "coming out the screen towards the user's eyes" — can be used in applications for sophisticated techniques, such as a pop-up panel.

Normally, you think of the Z axis within the scope of your activity and its widgets. However, there are ways to display "system alerts" – widgets that can float over the top of any activity. A `Toast` is the one you are familiar with, most likely. A `Toast` displays something on the screen, yet touch events on the `Toast` itself will be passed through to the underlying activity. Lookout demonstrated that it is possible to create a fully-transparent `Toast`. However, the lifetime of a `Toast` is limited (3.5 seconds maximum), which would limit how long it can try to grab touch events.

However, any application holding the `SYSTEM_ALERT_WINDOW` permission can display their own "system alerts" with custom look and custom duration. By making one that is fully transparent and lives as long as possible, a tapjacker can obtain touch events for any application in the system, including lock screens, home screens, and any standard activity.

Enter the Jackalope

To demonstrate this, let's take a look at the Jackalope sample application. It consists of a tiny activity and a service, with the service doing most of the work.

The activity employs `Theme.NoDisplay`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      android:versionCode="1"
      android:versionName="1.0" package="com.commonsware.android.tj.jackalope">
    <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW" />
    <application android:label="Jackalope">
        <activity android:name=".Jackalope"
                  android:theme="@android:style/Theme.NoDisplay">
            <intent-filter>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".Tapjacker" />
    </application>
</manifest>
```

The activity then just starts up the service and finishes:

```
package com.commonsware.android.tj.jackalope;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

public class Jackalope extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    startService(new Intent(this, Tapjacker.class));
    finish();
  }
}
```

The visible effect is... nothing. Tapping the icon in the launcher appears to have no effect, but it does actually start up the tapjacker. You just cannot see it.

The `Tapjacker` service does its evil work in a handful of lines of code:

```
package com.commonsware.android.tj.jackalope;

import android.app.Service;
import android.content.Intent;
import android.graphics.PixelFormat;
import android.os.IBinder;
import android.util.Log;
import android.view.Gravity;
import android.view.MotionEvent;
import android.view.View;
import android.view.WindowManager;

public class Tapjacker extends Service implements View.OnTouchListener {
  private View v=null;
  private WindowManager mgr=null;

  @Override
  public void onCreate() {
    super.onCreate();

    v=new View(this);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    v.setOnTouchListener(this);
    mgr=(WindowManager)getSystemService(WINDOW_SERVICE);

    WindowManager.LayoutParams params
      =new WindowManager.LayoutParams(
        WindowManager.LayoutParams.FILL_PARENT,
        WindowManager.LayoutParams.FILL_PARENT,
        WindowManager.LayoutParams.TYPE_SYSTEM_OVERLAY,
        WindowManager.LayoutParams.FLAG_WATCH_OUTSIDE_TOUCH,
        PixelFormat.TRANSPARENT);

    params.gravity=Gravity.FILL_HORIZONTAL|Gravity.FILL_VERTICAL;
    mgr.addView(v, params);

    // stopSelf(); -- uncomment for "component-less" operation
  }

  @Override
  public IBinder onBind(Intent intent) {
    return(null);
  }

  @Override
  public void onDestroy() {
    mgr.removeView(v);  // comment out for "component-less" operation

    super.onDestroy();
  }

  public boolean onTouch(View v, MotionEvent event) {
    Log.w("Tapjacker",
          String.valueOf(event.getX())+":"+String.valueOf(event.getY()));

    return(false);
  }
}
```

In `onCreate()`, we create an invisible `View` in Java code. Note that while you normally create a widget by passing in the Activity to the constructor, any Context will work, and so here we use the `Tapjacker` service itself.

Then, we access the `WindowManager` system service and add the invisible `View` to the system. To do this, we need to supply a `WindowManager.LayoutParams` object, much like you might use `LinearLayout.LayoutParams` or `RelativeLayout.LayoutParams` when putting a `View` inside of one of those containers. In this case, we:

1. Say that the `View` is to fill the screen
2. Indicates that the `View` is to be treated as a "system overlay" (`TYPE_SYSTEM_OVERLAY`), which will be at the top of the Z axis, floating above anything else (activities, dialogs, etc.)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

3. Indicates that we are to receive touch events that are beyond the `View` itself (`FLAG_WATCH_OUTSIDE_TOUCH`), such as on the system bar in API Level 11+ devices

We attach the `Tapjacker` service itself as the `OnTouchListener` to the `View`, and simply log all touch events to LogCat. In `onDestroy()`, we remove the system overlay `View`.

The result is that every screen tap results in an entry in LogCat – including data entry via the soft keyboard — even though the user is unaware that anything might be intercepting these events.

Note, though, that this does not intercept regular key events, including those from hardware keyboards. Also note that this does not magically give the malware author access to data entered before the tapjacker was set up. Hence, even if the tapjacker can sniff a password, if they do not know the account name, the user may still be safe.

Thinking Like a Malware Author

So, you have touch events. On the surface, this might not seem terribly useful, since the View cannot see what is being tapped upon.

However, a savvy malware author would identify what activity is in the foreground and log that information along with the tap details and the screen size, periodically dumping that information to some server. The malware author can then scan the touch event dumps to see what interesting applications are showing up. With a minor investment – and possibly collaboration with other malware authors — the author can know what touch events correspond to what keys on various input method editors, including the stock keyboards used by a variety of devices. Loading a pirated version of the APK on an emulator can indicate which activity has the password, PIN, or other secure data. Then, it is merely a matter of identifying the touch events applied to that activity and matching them up with the soft keyboard to determine what the user has entered. Over time, the malware author can perhaps develop a script to help automate this conversion.

Hence, the on-device tapjacker does not have to be very sophisticated, other than trying to avoid detection by the user. All of the real work to leverage the intercepted touch events can be handled offline.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Detecting Potential Tapjackers

Tapjacking seems bad.

This raises the question: can we identify when a tapjacker is running? That would allow users and developers to "route around the damage", such as uninstalling the tapjacker application.

Unfortunately, this does not appear to be possible. There is no obvious way for an application — or the user — to determine if some other application has employed `WindowManager` to add a `TYPE_SYSTEM_OVERLAY` `View` to the screen. Even if there were, there is no way to determine if this `View` represents a tapjacker or somebody exploiting this capability for other, less nefarious ends.

All we can do is identify applications that might pose a problem.

Who Holds a Permission?

The biggest identifier of a possible tapjacker is the `SYSTEM_ALERT_WINDOW` permission. This is required to add a `TYPE_SYSTEM_OVERLAY` `View` to the screen. Relatively few applications request this, since built-in system alerts, like `Toast`, do not require the permission.

Also, a tapjacker probably needs the `INTERNET` permission, to deliver the results to the malware author. In principle, the tapjacker could be split into two applications, one with `SYSTEM_ALERT_WINDOW` and one with `INTERNET`. However, this adds to deployment complexity and therefore may be avoided by malware authors.

An end user can use programs like RL Permissions to examine the applications that have these permissions. A developer can use `PackageManager` to enumerate the installed applications and see which ones hold these permissions. We will examine some code for doing this later in this chapter.

Who is Running?

Of course, a tapjacker is only a threat if it is actually running in the background. Applications might use those two permissions just in the course of normal activity-centric operations, not with an everlasting service trying to maintain the interception `View`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We can use `ActivityManager` to enumerate the running processes and what packages' code are in each. Any package that holds the permission combination from the previous section and is running in a process is a possible tapjacking threat. We will examine some code for doing this in the next section.

Note that it is important to examine running processes, not running services. For example, the `Tapjacker` service from earlier in this chapter could add the interception `View` and immediately exit. You can see this in action by adjusting the code as indicated in the comments in `onCreate()` and `onDestroy()`. The interception `View` will remain intact (with the `Tapjacker` service object leaked) until the process is terminated. That process might be terminated quickly or slowly, depending on what all is going on with the device. A sophisticated malware author might try to run without a running service to increase stealthiness, at the cost of occasionally losing some data.

Combining the Two: TJDetect

To see these techniques in action, take a look at the `Tapjacking/TJDetect` sample project. This consists of a single `ListActivity`, whose list is populated with the applications that hold both `SYSTEM_ALERT_WINDOW` and `INTERNET` permissions and are presently running:

```
package com.commonsware.android.tj.detect;

import android.app.ActivityManager;
import android.app.ListActivity;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import java.util.ArrayList;
import java.util.HashSet;

public class TJDetect extends ListActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    ActivityManager am=(ActivityManager)getSystemService(ACTIVITY_SERVICE);
    HashSet<CharSequence> runningPackages=new HashSet<CharSequence>();

    for (ActivityManager.RunningAppProcessInfo proc :
         am.getRunningAppProcesses()) {
      for (String pkgName : proc.pkgList) {
        runningPackages.add(pkgName);
      }
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    PackageManager mgr=getPackageManager();
    ArrayList<CharSequence> scary=new ArrayList<CharSequence>();

    for (PackageInfo pkg :
         mgr.getInstalledPackages(PackageManager.GET_PERMISSIONS)) {
      if (PackageManager.PERMISSION_GRANTED==
            mgr.checkPermission(android.Manifest.permission.SYSTEM_ALERT_WINDOW,
                                pkg.packageName)) {
        if (PackageManager.PERMISSION_GRANTED==
              mgr.checkPermission(android.Manifest.permission.INTERNET,
                                  pkg.packageName)) {
          if (runningPackages.contains(pkg.packageName)) {
            scary.add(mgr.getApplicationLabel(pkg.applicationInfo));
          }
        }
      }
    }

    setListAdapter(new ArrayAdapter(this,
                        android.R.layout.simple_list_item_1,
                        scary));
  }
}
```

To find the unique set of packages that are running across all processes, we iterate over the `RunningAppProcessInfo` objects returned by `ActivityManager` from a call to `getRunningAppProcesses()`. One public data member of `RunningAppProcessInfo` is a list of all the packages whose code runs in this process (`pkgList`). We use a simple `HashSet` to come up with the unique set of packages.

Then, we find all installed packages via a call to `getInstalledPackages()` on `PackageManager`. For each package, we use `checkPermission()` on `PackageManager` to see if the package in question holds a permission. Packages that pass those two tests are then checked against the `HashSet` of running packages, and those that are running are recorded in an `ArrayList`, later wrapped in an `ArrayAdapter`.

If you run TJDetect, it will not detect Jackalope, since Jackalope lacks the `INTERNET` permission. And, particularly on production hardware, it will detect several packages that may not be tapjackers at all, but rather are system applications installed in the firmware by the device manufacturer.

Defending Against Tapjackers

OK, so users and developers cannot reliably detect tapjackers. And Android 4.0.3 eliminates this attack vector. Surely, for previous versions of Android, there must be

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

something in the OS that helps defend users and developers against tapjacking, right?

The answer is "yes", for a generous definition of the term "defend" and an equally generous definition of "users and developers".

Filtering Touch Events

The only "defense" directly provided by Android is to allow applications to filter out touch events that had been intercepted by a tapjacker, `Toast`, or any other form of system overlay or alert. Those touch events are simply dropped, never delivered to the underlying activity.

Implementing the Filter

The simplest way to implement the touch event filter is to add the `android:filterTouchesWhenObscured` attribute to a widget or container, setting it to true. The equivalent Java setter method on `View` is `setFilterTouchesWhenObscured()`.

For example, take a look at the `res/layout/main.xml` file in the Tapjacking/RelativeSecure sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:filterTouchesWhenObscured="true">
  <TextView android:id="@+id/label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="URL:"
    android:layout_alignBaseline="@+id/entry"
    android:layout_alignParentLeft="true"/>
  <EditText
    android:id="@id/entry"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_toRightOf="@id/label"
    android:layout_alignParentTop="true"/>
  <Button
    android:id="@+id/ok"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/entry"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:layout_alignRight="@id/entry"
    android:text="OK" />
  <Button
    android:id="@+id/cancel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_toLeftOf="@id/ok"
    android:layout_alignTop="@id/ok"
    android:text="Cancel" />
</RelativeLayout>
```

Here, we have `android:filterTouchesWhenObscured="true"` on the `RelativeLayout` at the root of the layout resource. This property cascades to a container's children, and so if a tapjacker (or Toast or whatever) is above any of the widgets in the `RelativeLayout`, none of the touch events will be processed.

More fine-grained control can be achieved in custom widgets by overriding `onFilterTouchEventForSecurity()`, which gets control before the regular touch event methods. You can determine if a touch event had been intercepted by looking for the `FLAG_WINDOW_IS_OBSCURED` flag in the `MotionEvent` passed to `onFilterTouchEventForSecurity()`, and you can make the decision of how to handle this on an event-by-event basis.

The User Experience and the Hoped-For Security

Normally, the user will not see a difference when interacting with widgets that have this attribute set. However, if a tapjacker is intercepting these events, the user will not see any reaction from the widgets when they are tapped. For example, clicking a `Button` will have no visual effect (e.g., orange flash).

The hope is that users will realize that the UI is not responding to their touch events and therefore will not complete whatever it is they are doing. For example, they might not complete their PIN entry after realizing that the number pad supplied by the app is not responding to their taps.

For some users and some apps, this will be an effective defense. However, there will be some users who will remain oblivious until after completing the attempt to enter the private information.

The Flaws

The user can still use the soft keyboard to enter data into `EditText` widgets. While the soft keyboard will not automatically appear in portrait mode (since the `EditText`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

will not respond to the tap), if it has the focus, the user can long-press the MENU button to raise the soft keyboard and enter data that way.

Similarly, if the user is in landscape mode and gets the full-screen soft keyboard, since this is not the `EditText` widget defended by the touch event filtering, everything works normally — including interception by tapjacking. Developers could try to prevent this by adding `flagNoFullscreen` to the `android:imeOptions` attribute on the `EditText` in the layout XML, though this may not be honored by all soft keyboards. Developers could also try to prevent this by locking the activity into portrait mode (`android:screenOrientation="portrait"`), but this would be bad for users with side-slider keyboards, Android TV devices, etc.

And, most importantly, the tapjacking still happens. If users keep trying to enter their credentials despite the lack of UI feedback, they may eventually enter the whole thing and therefore become vulnerable to having that information used for ill ends.

Availability

Filtering touch events when the activity is obscured is supported in API Level 9 and above — in other words, Android 2.3 and newer. At the time of this writing, that leaves out ~25% of active Android devices, based on the June 1, 2012 published edition of the platform versions data from Google.

Detect-and-Warn

You can use the tapjacker detection logic illustrated earlier in this chapter. It is not particularly accurate, but you may feel it is worthwhile.

To minimize hassle for the user, your application should maintain a "whitelist" of approved packages. Any time you detect a package that is not on the approved list, you would raise an `AlertDialog` (or the equivalent) to let the user know of the potential tapjacker. If they elect to continue onward in your app, add the new package(s) to the whitelist, so you do not bother the user again for the same package.

Why Is This Being Discussed?

Some of you are by this time wondering why this book has a chapter on this subject.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Google's security team indicated to the author that `android:filterTouchesWhenObscured` is sufficient security. If so, developers need to realize when to use it, and for that, developers need to understand what tapjacking is to start with. The code to implement tapjacking is sufficiently trivial that "security by obscurity" of the code seems pointless.

It is eminently possible that `android:filterTouchesWhenObscured` is not sufficient security, despite Google's claim. Since Google seems to have changed their mind, eliminating tapjacking in Android 4.0.3, it would appear that Google thinks that Google's original solution was insufficient. In that case, developers may be able to help inform the public about the dangers of applications that request the `SYSTEM_ALERT_WINDOW` permission.

There are legitimate uses for tapjacking techniques. Some apps use this to provide a universal gesture interface, for example, to get control no matter what application is presently in the foreground. Whether the value that such apps provide is worth the risks inherent in tapjacking is up for debate.

If you feel that tapjacking is a problem and that `android:filterTouchesWhenObscured` is inadequate, you may wish to let Google know when you have the opportunity to interact with Google engineers at conferences and similar events. If you come up with other ways to detect and/or prevent tapjacking, you may wish to distribute that knowledge, so other developers can learn from your discovery.

What Changed in 4.0.3?

As of Android 4.0.3, the tapjacking attack is no longer possible, at least through the techniques outlined in this chapter. A `View` of type `TYPE_SYSTEM_OVERLAY` cannot receive touch events.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Miscellaneous Security Techniques

This chapter outlines some additional security measures that you can consider for your applications that do not necessarily warrant a full chapter on their own at this time.

In other words, it's just a pile of interesting security stuff.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. In addition, you should review the app signing chapter if you are unfamiliar with the signing process. Also, one demo uses remote services, including using the binding pattern, which is also covered in this book.

Public Key Validation

We sign our apps with signing keys all the time. By default, we are signing with a so-called "debug signing key", created automatically by the build tools. For production, we sign with a different signing key. The primary use of that signing key is to determine equivalence of authorship:

- Is this APK, representing an upgrade to an already-installed app, signed by the same signing key that signed that app?
- Is this APK, that requests firmware-defined `signature`-level permissions, signed by the same signing key that signed the firmware?

However, as it turns out, information about the public key that signed an APK is visible to us, for our own APK as well as for any other APK on the device. We can

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

leverage that to help determine whether a given APK was signed by something we recognize. This goes above and beyond using Android's built-in signature-based defenses (e.g., using a custom `signature`-level permission).

Scenarios

There are several scenarios in which we might imagine that we could employ our own public key validation. How well the technique will work, though, depends on what we are checking and the nature of the attack we are defending against.

Checking Yourself

You might consider checking your own app's public key. After all, if your app is not signed with your production signing key, something very strange is going on, and the natural reaction is that "something strange" is unlikely to be a good thing for you.

However, there are some issues here.

First and foremost, checking your own signing key assumes that whatever caused you to *not* be signed by that key did not also modify your validation algorithm. For example, suppose that you validate your signing key to determine if somebody perhaps reverse-engineered and modified your app, perhaps to remove some license checks. This will only catch an attacker that removed the licensing checks and did not *also* remove your signature validation, or modify the validation to use the attacker's signing key. While it is possible that an attacker will modify one part but not another, it remains unclear how well this defense will work in practice.

Also, bear in mind that you, as a developer, may be opting into services that intentionally change your app's signature. Various providers will "wrap" your app, whether for interstitial ad banners or for quasi-DRM. There are three possible ways that they wrap your app:

1. They sign it with their signing key, which means that your runtime validation of the key will fail, as your app is now signed by their key, not yours. This is also *very* risky, as if for whatever reason you are no longer able to use their service (e.g., they go out of business), you may have difficulty in upgrading your app, as you will not have the right key to use.
2. They sign it with *your* signing key, either one that you upload, or one that they generate for you. In this case, your runtime public key validation logic

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

could still work. On the other hand, now this other firm is perfectly capable of upgrading your app, or shipping other apps, signed with your production signing key, and this has its own set of risks.

3. They allow you to download the "wrapped" app and have you sign it yourself with your own signing key. This is the best alternative from a security standpoint, but it is the most tedious, as now you have additional work to do to publish your app.

Checking Arbitrary Other Apps

What will tend to be more reliable is to check *other* applications' public keys. While they might have been cracked, it is unlikely that the same attacker also attacked *your* app, and so you can help detect problems in others.

For example, let us consider a specific scenario: a client-side JAR for integration to a third-party app.

This book outlines many forms of IPC, from content providers to remote services to broadcast `Intents`. If you are creating an app that offers such IPC endpoints, you may wish to consider also shipping a JAR to make using those endpoints a bit easier. You might create a library that handles all of the details of sending commands to your remote service, or you might create a library that provides a wrapper around the AIDL-generated Java proxy classes for remote binding.

Another thing such a JAR could do is check the integrity of your app. The JAR's code is in the client's app, not yours, and while your app might be cracked, the client's app might not. You could check the validity of the public key of your own app from the client's app, and fail if there is a detected problem.

This might be especially important depending upon the nature of the app and the JAR that is providing access to it. If the app is an app offering on-device payments (e.g., a Google Wallet sort of app), and the app offers an API for other apps to do payments, it is fairly important that those other apps can trust the payment app. By checking the public key, your JAR can help provide that level of trust... or at least ensure that nobody else has done something specifically to degrade that trust.

This is particularly important for avoiding device-hosted man-in-the-middle attacks on your IPC from client apps to your app. In an ideal world, you would only allow IPC via `signature`-level permissions, but that will not work in cases where third parties are writing the clients.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If your IPC is based upon a service (command pattern or binding pattern), if multiple service implementations all advertise the same `<intent-filter>`, Android needs to decide which service will handle the request. First, it will take into account the `android:priority` value on the `<intent-filter>` (even though this behavior is currently undocumented). For multiple services with the same priority (e.g., no priority specified), the first one that was installed will be the one that is chosen. In either case, the client has no way to know, short of examining the service's public key, whether the service that will respond to the requests for IPC is the legitimate service or something else advertising that it supports the same `Intent` action.

The Easy Solution: SignatureUtils

The author of this book has published the CWAC-Security library. Among other things, this library has a `SignatureUtils` class that makes it relatively easy for you to compare the signature of some Android app to a known good value.

All you need to do is call the static `getSignatureHash()` method, supplying some `Context` (any will do) and the package name of the app that you wish to check. This will return the SHA–256 hash of the signing key of the app, as a set of capitalized, colon-delimited hex values.

You can get the same sort of hash by running the Java 7 version of `keytool`. Hence, if the app you wish to test is another one of yours, perhaps signed with a different signing key, you can use `keytool` to get the value to compare with the result of `getSignatureHash()`. Or, during development, create a little utility app that will dump the `getSignatureHash()` value for the third-party app, and run it on a device containing a known good version of that app (i.e., one that does not appear to have been replaced by malware).

Ideally, over time, we will be able to get app developers to publish their SHA–256 hashes on their Web sites, as another means of getting a known value of the hash to compare at runtime.

If you determine that `getSignatureHash()` does *not* return the right value, this means that the app that is installed on the device is written by somebody other than the app's original author. Often times, this will mean the app has malware in it. It is up to you to determine how you wish to respond to this scenario:

- Alert the user?
- Send data back to your server, or to your analytics collection point, with details of the bad APK?

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Block usage of your app, or usage of features that depend upon the flawed third party?
- Something else?

Examining Public Keys

Under the covers, `SignatureUtils` uses `PackageManager` and related classes to examine what they somewhat erroneously refer to as "signatures". The `MiscSecurity/SigDump` sample project will allow us to browse the list of installed packages, see a decoded public key on the screen for a package that we select, plus dump the "signature" as a binary file for later comparison using another app.

The UI Structure

In this sample, we use a `SlidingPaneLayout` for a master-detail pattern presentation, as was demonstrated in the chapter on dealing with multiple screen sizes. The "master" fragment will be the list of packages; the "detail" fragment will be the decoded public key for the selected package.

The master fragment is implemented as `PackagesFragment`. It implements a typical `ListFragment` for use with the master/detail pattern, utilizing the `activated` state to show the context for the detail fragment. The detail will be `SignatureFragment`, which will display portions of the decoded public key in a `TableLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:shrinkColumns="1"
  android:stretchColumns="1">

  <TableRow>

    <TextView
      android:layout_gravity="center"
      android:layout_margin="4dp"
      android:text="@string/subject"
      android:textStyle="bold"/>

    <TextView android:id="@+id/subject"/>
  </TableRow>

  <TableRow>

    <TextView
      android:layout_gravity="center"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      android:layout_margin="4dp"
      android:text="@string/issuer"
      android:textStyle="bold"/>

    <TextView android:id="@+id/issuer"/>
  </TableRow>

  <TableRow>

    <TextView
      android:layout_gravity="center"
      android:layout_margin="4dp"
      android:text="@string/valid_between"
      android:textStyle="bold"/>

    <TextView android:id="@+id/valid"/>
  </TableRow>

</TableLayout>
```

Listing the Packages

Our `PackagesFragment` needs the list of packages to display. It expects the hosting activity to supply that, by using the contract pattern, and having a `getPackageList()` method on its `Contract`:

```
  interface Contract {
    void onPackageSelected(PackageInfo pkgInfo);

    List<PackageInfo> getPackageList();
  }
```

The hosting activity — `MainActivity` — retrieves a `PackageManager` instance in `onCreate()`, caching it in a `mgr` data member. `getPackageList()` then calls `getInstalledPackages()` on `PackageManager`, specifically requesting to retrieve signature information via the `GET_SIGNATURES` flag. The list we get back from `getInstalledPackages()` can be in any order, so we sort the results before returning it for display purposes:

```
  @Override
  public List<PackageInfo> getPackageList() {
    List<PackageInfo> result=
        mgr.getInstalledPackages(PackageManager.GET_SIGNATURES);

    Collections.sort(result, new Comparator<PackageInfo>() {
      @Override
      public int compare(final PackageInfo a, final PackageInfo b) {
        return(a.packageName.compareTo(b.packageName));
      }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    });

    return(result);
  }
```

Note that this is a `List` of `PackageInfo` objects, so we need an `ArrayAdapter` subclass to handle rendering that. Here, we have a `PackageListAdapter` that knows how to populate list rows using the `packageName` field of a `PackageInfo` object, plus using an activated row layout for API Level 11+ devices:

```
package com.commonsware.android.signature.dump;

import android.annotation.TargetApi;
import android.content.pm.PackageInfo;
import android.os.Build;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.TextView;

class PackageListAdapter extends ArrayAdapter<PackageInfo> {
  PackageListAdapter(PackagesFragment packagesFragment) {
    super(packagesFragment.getActivity(), getRowResourceId(),
          packagesFragment.getContract().getPackageList());
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    View result=super.getView(position, convertView, parent);

    ((TextView)result).setText(getItem(position).packageName);

    return(result);
  }

  @TargetApi(Build.VERSION_CODES.HONEYCOMB)
  private static int getRowResourceId() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
      return(android.R.layout.simple_list_item_activated_1);
    }

    return(android.R.layout.simple_list_item_1);
  }
}
```

The result is that our master list is a list of all installed packages, sorted by package name, with the detail `TableLayout` peeking out of the right edge when shown on a phone-sized screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

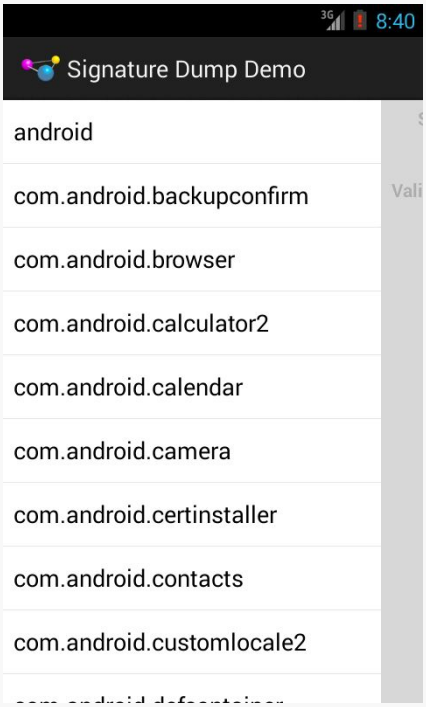


Figure 664: Signature Dump Demo, As Initially Launched

Dumping the Key

`onListItemClick()` of our `PackagesFragment` routes control to `onPackageSelected()` of the `Contract` interface, which in our case is `MainActivity`. There, we need to do some useful stuff based upon the fact that the user tapped on a particular package:

```
@Override
public void onPackageSelected(PackageInfo pkgInfo) {
  Signature[] signatures=pkgInfo.signatures;
  byte[] raw=signatures[0].toByteArray();

  sigDisplay.show(raw);
  panes.closePane();

  File output=
      new File(getExternalFilesDir(null),
               pkgInfo.packageName.replace('.', '_') + ".bin");

  new WriteThread(output, raw).start();
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

First, we get the `Signature` array from the `PackageInfo` object. While this is an array, usually an app will only be signed once. Signing more than once is not especially useful, as an upgraded app needs to match the count and contents of each signature. Hence, we will only pay attention to the first signature. If you are using these techniques as the basis for your client JAR checking the public key of your app for IPC protection purposes, *and* your app is signed with multiple keys, you will want to check all of those keys.

The public key itself is represented as a byte array in the `Signature`. `onPackageSelected()` does two things with this byte array:

- Writes it to a file on external storage using a background thread, with a filename based on the app's package name, with `.` characters replaced by `_` characters
- Passes the byte array to the detail fragment (a `SignatureFragment`) and updates the `SlidingPaneLayout` to ensure that detail fragment is now visible

Decoding the Key

`SignatureFragment` is mostly comprised of the `show()` method that `MainActivity` uses to pass us the byte array of the "signature" to display:

```
package com.commonsware.android.signature.dump;

import android.app.Fragment;
import android.os.Bundle;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;
import java.io.ByteArrayInputStream;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Locale;

public class SignatureFragment extends Fragment {
  DateFormat fmt=new SimpleDateFormat("yyyy-MM-dd HH:mm:ss", Locale.US);

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    return(inflater.inflate(R.layout.sig, container, false));
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }

  void show(byte[] raw) {
    CertificateFactory cf=null;

    try {
      cf=CertificateFactory.getInstance("X509");
    }
    catch (CertificateException e) {
      Log.e(getClass().getSimpleName(),
            "Exception getting CertificateFactory", e);
      return;
    }

    X509Certificate c=null;
    ByteArrayInputStream bin=new ByteArrayInputStream(raw);

    try {
      c=(X509Certificate)cf.generateCertificate(bin);
    }
    catch (CertificateException e) {
      Log.e(getClass().getSimpleName(),
            "Exception getting X509Certificate", e);
      return;
    }

    TextView tv=(TextView)getView().findViewById(R.id.subject);

    tv.setText(c.getSubjectDN().toString());

    tv=(TextView)getView().findViewById(R.id.issuer);
    tv.setText(c.getIssuerDN().toString());

    tv=(TextView)getView().findViewById(R.id.valid);
    tv.setText(fmt.format(c.getNotBefore()) + " to "
        + fmt.format(c.getNotAfter()));
  }
}
```

That byte array *really* represents an X509 certificate, serialized to a byte array. `show()` goes through the work to get the `X509Certificate` object representing that same data, assuming the byte array is not corrupted somehow. Then, `show()` populates some `TextView` widgets in our `TableLayout` with the:

- The subject of the signature
- The issuer of the signature
- The range of dates in which this signature is valid

A debug signing key output will resemble:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 665: Signature Dump Demo, Showing Debug Signing Key

A self-signed production signing key will resemble:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

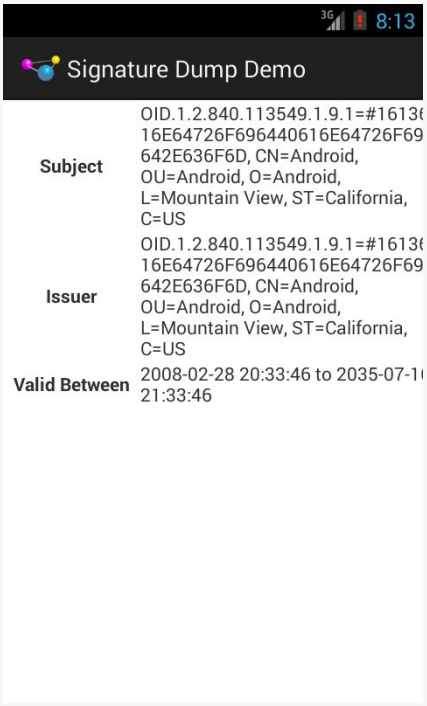


Figure 666: Signature Dump Demo, Showing Production Signing Key

A signing key created by some signing authority would have a subject that is distinct from its issuer.

Validating a Service's Public Key

Seeing these public keys is one thing — using them to help protect the user from man-in-the-middle attacks is another.

In this section, we will look at how to validate the public key of a service that is exporting some API that a client app will use, so the client app knows that it is talking to the right service implementation. There are three sample projects that make up this demonstration, all found in the `MiscSecurity/SecureServiceIPC` directory of the GitHub repository.

The Services

The demo contains two separate apps, each implementing a `Service`. The `GoodService` project has `GoodService`, which contains support for both the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

command and binding patterns, where the only effective behavior for either of them is to log positive-sounding messages to LogCat:

```
package com.commonsware.android.secsvc.demo1;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.RemoteException;
import android.util.Log;
import com.commonsware.android.secsvc.SomethingUseful;

public class GoodService extends Service {

  @Override
  public IBinder onBind(Intent arg0) {
    return(new GoodBinder());
  }

  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    Log.d(getClass().getSimpleName(), "It's all good");

    stopSelf();

    return(START_NOT_STICKY);
  }

  static class GoodBinder extends SomethingUseful.Stub {
    @Override
    public void hi() throws RemoteException {
      Log.d(getClass().getSimpleName(), "It's all good");
    }
  }
}
```

The `BadService` project has `BadService`, with the same implementation, except that the logging messages indicate that the service is, in fact, evil.

However, *both* projects have their services defined in the manifest with the same `<intent-filter>`, tying each of their services to the `com.commonsware.android.secsvc.SOMETHING` custom action:

```
    <service
      android:name="GoodService"
      tools:ignore="ExportedService">

      <intent-filter>
        <action android:name="com.commonsware.android.secsvc.SOMETHING"/>
      </intent-filter>
    </service>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The difference is that the `BadService` adds a high `android:priority` value, in an attempt to masquerade as the `GoodService` and convince clients to talk to it instead:

```
    <service
      android:name="BadService"
      tools:ignore="ExportedService">
      <intent-filter android:priority="2147483647">
        <action android:name="com.commonsware.android.secsvc.SOMETHING"/>
      </intent-filter>
    </service>
```

Using the Services

The third project in the demo, `Client`, has a `MainActivity` with a layout consisting of two `Button` widgets and a `Switch`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  tools:context=".MainActivity">

  <Switch
    android:id="@+id/verify"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="4dip"
    android:text="@string/enable_verification"/>

  <Button
    android:id="@+id/command"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_margin="4dip"
    android:layout_weight="1"
    android:onClick="testCommand"
    android:text="@string/test_command"/>

  <Button
    android:id="@+id/binding"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_margin="4dip"
    android:layout_weight="1"
    android:onClick="testBinding"
    android:text="@string/test_binding"/>

</LinearLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Not surprisingly, the "Test Command" button sends a command to the `com.commonsware.android.secsvc.SOMETHING` service, while the "Test Binding" button will bind to the service, call the `hi()` method of the service's AIDL-defined API, and then unbind from the service.

And, by default, the `Intent` used for sending the command or initiating the binding is simply one that identifies the `com.commonsware.android.secsvc.SOMETHING` action string:

```
  private static final String ACTION_SERVICE=
      "com.commonsware.android.secsvc.SOMETHING";
  private static final Intent INTENT_SERVICE=new Intent(ACTION_SERVICE);
```

Validating the Service

If the `Switch` is checked, though, we want to validate that the service is the right one for our activity to talk to.

The `buildServiceIntent()` method in `MainActivity` creates the `Intent` that we will use for sending the command or binding to the service. In there, if the `verify Switch` is checked, we iterate over all possible services that advertise that they support our custom action string:

```
  Intent buildServiceIntent() {
    Intent result=null;

    if (verify.isChecked()) {
      PackageManager mgr=getPackageManager();

      for (ResolveInfo info : mgr.queryIntentServices(INTENT_SERVICE, 0)) {
        try {
          if (validate(info.serviceInfo.packageName,
                       R.raw.valid_signature)) {
            result=new Intent(INTENT_SERVICE);
            result.setComponent(new ComponentName(
                                                  info.serviceInfo.packageName,
                                                  info.serviceInfo.name));

            break;
          }
        }
        catch (Exception e) {
          Log.e(getClass().getSimpleName(),
                "Exception finding valid service", e);
        }
      }
    }
    else {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      result=INTENT_SERVICE;
    }

    return(result);
  }
```

Probably `queryIntentServices()` will return a list of size `1`. If it returns an empty list, `buildServiceIntent()` will return `null`, and the caller can know to avoid trying to work with the service. However, `buildServiceIntent()` will *also* return `null` if there is no valid service to talk to — even if `queryIntentServices()` indicates that there is a matching `Service`. That is because `buildServiceIntent()` calls `validate()` to compare the public key of the service with a known good edition of that key. Here, we package that "known good edition" as a raw resource, named `valid_signature`. This is the signature dumped by the `SigDump` sample application described earlier in this chapter.

`validate()`, in turn, finds the `signature` of the service's package and compares it with the contents of the raw resource:

```
  boolean validate(String pkg, int raw) throws NameNotFoundException,
                                               NotFoundException, IOException {
    PackageManager mgr=getPackageManager();
    PackageInfo pkgInfo=
        mgr.getPackageInfo(pkg, PackageManager.GET_SIGNATURES);
    Signature[] signatures=pkgInfo.signatures;
    byte[] local=signatures[0].toByteArray();

    return(isEqual(new ByteArrayInputStream(local),
                   getResources().openRawResource(raw)));
  }
```

The `isEqual()` method is simply some code, found in a Stack Overflow answer, that compares the bytes of two streams, looking for a match:

```
  // from http://stackoverflow.com/a/4245881/115145

  private boolean isEqual(InputStream i1, InputStream i2)
                                                          throws IOException {
    byte[] buf1=new byte[1024];
    byte[] buf2=new byte[1024];

    try {
      DataInputStream d2=new DataInputStream(i2);
      int len;
      while ((len=i1.read(buf1)) >= 0) {
        d2.readFully(buf2, 0, len);
        for (int i=0; i < len; i++)
          if (buf1[i] != buf2[i])
            return false;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      }
      return d2.read() < 0; // is the end of the second file
                            // also.
    }
    catch (EOFException ioe) {
      return false;
    }
    finally {
      i1.close();
      i2.close();
    }
  }
```

The `buildServiceIntent()` method not only calls `validate()`, but then calls `setComponent()` on the `Intent` for the service with the matching public key. This ensures that we will *only* talk to this `Service` implementation, and not any others.

If you wish to test this behavior, you will need to have the `GoodService` and the `BadService` apps signed with different signing keys, and use the `SigDump` sample app to get the binary representation of that signing key to use in your client app.

What you will find is that if validation is enabled, the client app will only work with `GoodService`, even if `BadService` is installed. Without this validation, if `BadService` is installed along with `GoodService`, the client app will work with `BadService`, due to the higher `android:priority` on the `BadService`.

You might argue that, in this case, we could simply see if the package name is the right one, rather than fussing around with the public keys. That is true... in this case. The more likely attack, though, is that the service app is replaced by a modified version of the same service app, with a bit of additional code that makes use of the data flowing between client and service. In that case, the package name, and even the class name of the service, are likely to be unmodified. However, unless the real services' author lost control of their production signing key, the attacker's app would be signed by a *different* signing key, making the public key validation important.

Choosing Your Signing Keysize

The documentation for app signing contains a small side note about the `-keysize` parameter to `keytool`, the utility used to generate our signing keys:

> The size of each generated key (bits). If not supplied, Keytool uses a default key size of 1024 bits. In general, we recommend using a key size of 2048 bits or higher.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The reason for the 2,048-bit key size recommendation is that 1,024-bit RSA (the `keytool` default) has been considered at risk for a few years.

The recent revelations about state-sponsored decryption research should be hammering this home. Even if today, forging a 1,024-bit digital signature is still impractical for all but the largest security agencies, it is well within reason that this will fall within the reach of large botnets in the not-too-distant future. Once signing keys can be cracked, apps will be able to be replaced with hacked editions, without tripping up the signature check, or `signature`-level permission checks might start passing due to forged signatures.

Switching to a larger keysize is not that hard... for new apps. Just specify `-keysize 4096` when creating your production signing key, and you should be good for a long time, barring a major decryption breakthrough for RSA signatures.

For existing apps with existing signing keys, though, you cannot change the key without breaking your ability to update the app.

Create a new, stronger production signing key, as a separate key from whatever you are using for production. Make note to use that new signing key for any new apps you create. And, if you have other reasons why you are migrating an existing user base to a new app (e.g., free app for which you are now offering a paid-app option), consider using the new signing key.

If you are a consultant, and you create unique signing keys per project, just cut over to using a stronger key for new clients and projects.

And if you are creating apps for which security is paramount, you might consider whether it is worthwhile to move your user base to a new version of the app with a new signing key at some point, just for the added protection.

Avoiding Accidental APIs

One place where developers create their own security problems is with "accidental APIs".

An API, of course, is where one code base exposes some interface that another code base can use. An accidental API is when one code base does not intend to expose an interface, but does anyway, possibly to the app's detriment.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Bear in mind that if your app becomes popular, other developers will poke and prod at it, to see if they can connect to your app by one means or another. Perhaps they want to offer features that you have not gotten to yet. Perhaps they have more nefarious aims. Regardless, making sure that other code can only work with your app the way that *you* intend for such code to work with your app.

Export Only What's Necessary

A component of your app is only reachable by a third-party app if it is exported. Otherwise, it is inaccessible to third-party apps.

(Admittedly, content providers have an exception to this rule, which we will get to shortly)

You normally do not think about exporting components, except when it comes to content providers. However, your choices for how you implement your app may lead you to accidentally export things that you did not realize were exported.

Export Defaults

The official way to declare whether or not a component is exported is to have an `android:exported` attribute for that component in the manifest (e.g., on an `<activity>` element). However, many times, we do not have such an attribute, but instead rely on the default export behavior.

Activities, services, and broadcast receivers have a simple rule for the default: if the component has an `<intent-filter>`, it is exported by default. Otherwise, it is not exported by default.

This, in turn, leads to a fairly simple development rule: only use an `<intent-filter>` and implicit `Intent`s for working with your components if you *also* want third party apps to work with those components. Otherwise, do not use `<intent-filter>`, and instead communicate with your components using explicit `Intent`s (e.g., the kind that take a Java class as the second constructor parameter).

For example, the classic `MAIN/LAUNCHER` `<intent-filter>` on your launcher activity is specifically there because you *want* a third party app — the launcher — to be able to start your activity. Most, if not all, of your other activities probably do not need an `<intent-filter>`, as they are likely to be private to your app.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Chooser Bug

Some developers choose to still use an `<intent-filter>` and implicit `Intents` for their own private activities, yet then use `android:exported` to enforce the privacy.

This is not a good plan.

The rest of the system, notably `PackageManager`, does not pay much attention to `android:exported` *until the time when the component is to be used*, such as when the activity is to be started. Then, and only then, does Android realize that the component is not exported, and it fails the request, usually with a cryptic `SecurityException`.

A classic example of where this can cause problem came to light in 2012, with the UPS Mobile app. The following is an excerpt from the author's blog post on this incident:

The UPS Mobile app allows you to track packages and do a handful of other things that you might ordinarily do via the UPS Web site. It generally seems to be well-regarded, but it has an annoying flaw:

It claims to be Barcode Scanner, and does a lousy job at it.

Barcode Scanner, from ZXing, is a favorite among Android developers for its integration possibilities. However, some people do not like having a dependence upon the Barcode Scanner app, so they grab the open source code and attempt to blend it into their own apps. This is neither endorsed nor supported by the ZXing team, but since it is open source, it is also perfectly legitimate.

However, UPS (or whoever they hired to build the app) screwed up. They not only copied the source code, but they copied the manifest entry for the scanning activity. And, their activity has:

```
<intent-filter>
  <action android:name="com.google.zxing.client.android.SCAN" />
  <category android:name="android.intent.category.DEFAULT" />
<intent-filter>
```

This means that on any device that has UPS Mobile installed, they will be an option for handling Barcode Scanner `Intents`. What happened was that the person asking the question was manually invoking `startActivityForResult()` to bring up

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Barcode Scanner, was getting a chooser with UPS Mobile in it, and then was crashing upon choosing UPS Mobile... because UPS Mobile declared this activity to be not exported.

Due to this bug, Android will display non-exported activities in a chooser, despite the fact that they can never be successfully used by the user.

So, what should we learn from this?

First, UPS Mobile should not have used that `<intent-filter>`. As Dianne Hackborn has pointed out, your `<intent-filter>` mix is effectively part of your app's API, and so you need to think long and hard about every `<intent-filter>` you publish. UPS Mobile is not Barcode Scanner and should not be advertising that they handle such `Intents`, despite the activity being not exported.

Second, UPS Mobile probably should not have had *any* `<intent-filter>` elements for this activity, if they intend to use it purely internally. They could just as easily use an explicit `Intent` to identify the activity and avoid all of this nonsense.

Third, the person who filed the SO question ideally would have been using ZXing's `IntentIntegrator`. As Sean Owen of the ZXing project noted in a comment on my answer, `IntentIntegrator` ensures that only Barcode Scanner or official brethren will handle any scan requests, so this problem would not have appeared.

Fourth, Android really should not be showing non-exported activities in a chooser, which means probably that `PackageManager` should be filtering out non-exported activities from methods like `queryIntentActivities()`, which I presume lies at the heart of the chooser.

In summary, if your component is truly private, do not have an `<intent-filter>` on it, lest you cause yourself, and your users, problems with other apps.

The ContentProvider Behavior Change

Content providers are a little different... in lots of ways. In the specific scenarios being covered here, there are two primary differences.

First, third-party apps *can* still access a provider that has `android:exported="false"`. However, they can only do so in response to some operation initiated by your application, using `android:grantUriPermissions` and

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

flags like FLAG_GRANT_READ_URI_PERMISSION. A third-party app will have no *independent* access to your non-exported provider.

Second, the default value for android:exported not only does not depend upon <intent-filter> (since few providers use one), but it has changed over the years:

- For apps with android:minSdkVersion *and* android:targetSdkVersion set to 16 or lower, the provider is exported by default
- All other apps, the provider is *not* exported by default

Lint will complain about your manifest having a <provider> without an android:exported attribute.

Sanitize Your Input Extras

If you do expose one or more of your components to third-party apps, and you are supporting certain Intent extras on any Intents used to talk to those components, make sure that the extras' values make sense.

Even Google makes this error, as was seen in the PreferenceActivity bug. PreferenceActivity supports an extra, named :android:show_fragment, to indicate that the activity should immediately jump to a specific fragment, rather than start at the top level of the preference navigation. The problem is that PreferenceActivity did not — and, at the time, could not — validate that the fragment to be loaded is a fragment that is *supposed* to be loaded. This would allow attackers to force apps, like Settings, to load arbitrary fragments, including those not normally accessible to the current user. This is the reason why we now need to override isValidFragment() in our PreferenceActivity implementations, so *we* can declare whether or not a particular requested fragment is a legitimate choice or not.

The equivalent behavior for a ContentProvider is to sanitize the inputs to methods like query(), update(), openFile(), and so on, to make sure that you do not expose something that you should not. For example, blindly accepting paths to openFile() could get you in trouble, if the Uri contains relative paths (e.g., content://your.authority.here/../databases/your-private.db), perhaps allowing third parties to get at files that you did not intend for them to access.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Secure Your Output Extras

Similarly, if you send broadcasts or otherwise use IPC to talk to third-party apps, bear in mind that others might be able to see some of that interaction, depending on the IPC in question.

The obvious case is with a broadcast `Intent` for an implicit `Intent`. Any app with a registered receiver will be able to "tune into" that broadcast and get whatever data is inside the `Intent`. In cases where you cannot use permissions to limit the scope of the broadcast, you need to make sure that there is nothing in the `Intent` that is private to the user.

Sometimes, though, non-obvious cases will emerge. For a few years, `Intent` extras on activities might be viewed by third-party apps that held the `GET_TASKS` permission, courtesy of the recent-tasks list. The `Intent` used to launch the task is available via `ActivityManager` and `getRecentTasks()`. While this specific problem was resolved in Android 4.1.1, there may be other similar scenarios lurking about.

Other Ways to Expose Data

Sometimes, we expose data to third-party apps by using standard Android APIs. We focus on the normal publisher and consumer of data using those APIs and forget about other apps that might be monitoring those communications. Or, we might not realize that one party in those communications may not have the user's best interests at heart. This section outlines some examples.

App Widgets

Any data that is put into the widgets inside of your `RemoteViews` for an app widget is visible to the home screen, lockscreen, or other app widget host. Those apps are the ones actually converting the `RemoteViews` into a view hierarchy, and they can inspect those views, reading the text in your `TextViews`, and so forth.

As a result, be careful about exposing potentially sensitive data via an app widget.

Notifications

Custom notifications also use `RemoteViews` and therefore could suffer from the same problem.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

On the surface, you might not be worried quite so much about this, because the `Notification` object goes to the `NotificationManager`, for display by the OS itself.

However, as of Android 4.3 (API Level 18), apps can register to listen to added and removed notifications via a `NotificationListenerService`. Not only can such a service read the text from your `Notification`, but it can also access your `RemoteViews`. This includes any `RemoteViews` that may be generated for you by the expanded notification classes (e.g., `BigPictureStyle`).

As a result, be careful about exposing potentially sensitive data via a `Notification`.

Clipboard

Any app can retrieve text off of the clipboard. After all, that's the point behind a clipboard.

However, this does mean that you need to be careful what you put on the clipboard in the first place. The quintessential problem case is a password manager: putting a password on the clipboard for easy pasting into an app's `EditText` password field will be popular, but it allows that password to be retrieved by other apps.

You can attempt to help reduce the window of risk by clearing the clipboard after a period of time. However, bear in mind that your process might be terminated before that occurs. Also, only clear the clipboard if the clipboard text is still yours — do not clear the clipboard if another app has already put its own contents there, lest you confuse and irritate the user in the middle of some other paste operation.

ServerSocket and Kin

If you open up any sort of server-style socket connection — TCP/IP, Bluetooth, etc. — bear in mind that the Android security framework may not be able to help you much. You cannot secure a `ServerSocket` with an `android:permission` attribute, for example. It is up to you to validate whether a particular request is expected and allowed, or not.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Hardware and System Services

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Accessing Location-Based Services

A popular feature on current-era mobile devices is GPS capability, so the device can tell you where you are at any point in time. While the most popular use of GPS service is mapping and directions, there are other things you can do if you know your location. For example, you might set up a dynamic chat application where the people you can chat with are based on physical location, so you are chatting with those you are nearest. Or, you could automatically "geotag" posts to Twitter or similar services.

GPS is not the only way a mobile device can identify your location. Alternatives include:

1. Cell tower triangulation, where your position is determined based on signal strength to nearby cell towers
2. Proximity to public WiFi "hotspots" that have known geographic locations
3. GPS alternatives, such as GLONASS (Russia), Galileo (European Union, still under development), and Compass (China, still under development)

Android devices may have one or more of these services available to them. You, as a developer, can ask the device for your location, plus details on what providers are available. There are even ways for you to simulate your location in the emulator, for use in testing your location-enabled applications.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on threads.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Location Providers: They Know Where You're Hiding

Android devices can have access to several different means of determining your location. Some will have better accuracy than others. Some may be free, while others may have a cost associated with them. Some may be able to tell you more than just your current position, such as your elevation over sea level, or your current speed.

Android, therefore, has abstracted all this out into a set of `LocationProvider` objects. Your Android environment will have zero or more `LocationProvider` instances, one for each distinct locating service that is available on the device. Providers know not only your location, but also their own characteristics, in terms of accuracy, cost, etc.

You, as a developer, will use a `LocationManager`, which holds the `LocationProvider` set, to figure out which `LocationProvider` is right for your particular circumstance. You will also need a permission in your application, or the various location APIs will fail due to a security violation. Depending on which location providers you wish to use, you may need `ACCESS_COARSE_LOCATION`, `ACCESS_FINE_LOCATION`, or both. Note that `ACCESS_COARSE_LOCATION` may intentionally filter out location fixes that are "too good" (i.e., more accurate than a city block).

Finding Yourself

The obvious thing to do with a location service is to figure out where you are right now.

To do that, you need to get a `LocationManager` — call `getSystemService(LOCATION_SERVICE)` from your activity or service and cast it to be a `LocationManager`.

The next step to find out where you are is to get the name of the `LocationProvider` you want to use. Here, you have two main options:

- Ask the user to pick a provider
- Find the best-match provider based on a set of criteria

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you want the user to pick a provider, calling `getProviders()` on the `LocationManager` will give you a `List` of providers, which you can then present to the user for selection.

Or, you can create and populate a `Criteria` object, stating the particulars of what you want out of a `LocationProvider`, such as:

1. `setAltitudeRequired()` to indicate if you need the current altitude or not
2. `setAccuracy()` to set a minimum level of accuracy, in meters, for the position
3. `setCostAllowed()` to control if the provider must be free or if it can incur a cost on behalf of the device user

Given a filled-in `Criteria` object, call `getBestProvider()` on your `LocationManager`, and Android will sift through the criteria and give you the best answer. Note that not all of your criteria may be met – all but the monetary cost criterion might be relaxed if nothing matches.

You are also welcome to hard-wire in a `LocationProvider` name (e.g., `GPS_PROVIDER`), perhaps just for testing purposes.

Once you know the name of the `LocationProvider`, you can call `getLastKnownLocation()` to find out where you were recently. However, unless something else is causing the desired provider to collect fixes (e.g., unless the GPS radio is on), `getLastKnownLocation()` will return `null`, indicating that there is no known position. On the other hand, `getLastKnownLocation()` incurs no monetary or power cost, since the provider does not need to be activated to get the value.

These methods return a `Location` object, which can give you the latitude and longitude of the device in degrees as a Java `double`. If the particular location provider offers other data, you can get at that as well:

1. For altitude, `hasAltitude()` will tell you if there is an altitude value, and `getAltitude()` will return the altitude in meters.
2. For bearing (i.e., compass-style direction), `hasBearing()` will tell you if there is a bearing available, and `getBearing()` will return it as degrees east of true north.
3. For speed, `hasSpeed()` will tell you if the speed is known and `getSpeed()` will return the speed in meters per second.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A more likely approach to getting the `Location` from a `LocationProvider`, though, is to register for updates, as described in the next section.

On the Move

Not all location providers are necessarily immediately responsive. GPS, for example, requires activating a radio and getting a fix from the satellites before you get a location. That is why Android does not offer a `getMeMyCurrentLocationNow()` method. Combine that with the fact that your users may well want their movements to be reflected in your application, and you are probably best off registering for location updates and using that as your means of getting the current location.

The `Internet/Weather` sample application shows how to register for updates — call `requestLocationUpdates()` on your `LocationManager` instance. This takes four parameters:

- The name of the location provider you wish to use
- How long, in milliseconds, *should* have elapsed before we might get a location update
- How far, in meters, must the device have moved before we might get a location update
- An implementation of the `LocationListener` interface that will be notified of key location-related events

`LocationListener` requires four methods, the big one being `onLocationChanged()`, where you will receive your `Location` object when an update is ready:

```
@Override
public void onLocationChanged(Location location) {
  FetchForecastTask task=new FetchForecastTask();

  task.execute(location);
}
```

Bear in mind that the time parameter is only a guide to help steer Android from a power consumption standpoint. You may get many more location updates than this. To get the maximum number of location updates, supply `0` for both the time and distance constraints.

When you no longer need the updates, call `removeUpdates()` with the `LocationListener` you registered. If you fail to do this, your application will

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

continue receiving location updates even after all activities and such are closed up, which will also prevent Android from reclaiming your application's memory.

There is another version of `requestLocationUpdates()` that takes a `PendingIntent` rather than a `LocationListener`. This is useful if you want to be notified of changes in your position even when your code is not running. For example, if you are logging movements, you could use a `PendingIntent` that triggers a `BroadcastReceiver` (`getBroadcast()`) and have the `BroadcastReceiver` add the entry to the log. This way, your code is only in memory when the position changes, so you do not tie up system resources while the device is not moving.

Are We There Yet? Are We There Yet? Are We There Yet?

Sometimes, you want to know not where you are now, or even when you move, but when you get to where you are going. This could be an end destination, or it could be getting to the next step on a set of directions, so you can give the user the next turn.

To accomplish this, `LocationManager` offers `addProximityAlert()`. This registers a `PendingIntent`, which will be fired off when the device gets within a certain distance of a certain location. The `addProximityAlert()` method takes, as parameters:

1. The latitude and longitude of the position that you are interested in
2. A radius, specifying how close you should be to that position for the `Intent` to be raised
3. A duration for the registration, in milliseconds — after this period, the registration automatically lapses. A value of `-1` means the registration lasts until you manually remove it via `removeProximityAlert()`.
4. The `PendingIntent` to be raised when the device is within the "target zone" expressed by the position and radius

Note that it is not guaranteed that you will actually receive an `Intent`, if there is an interruption in location services, or if the device is not in the target zone during the period of time the proximity alert is active. For example, if the position is off by a bit, and the radius is a little too tight, the device might only skirt the edge of the target zone, or go by so quickly that the device's location isn't sampled while in the target zone.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It is up to you to arrange for an activity or receiver to respond to the `Intent` you register with the proximity alert. What you then do when the `Intent` arrives is up to you: set up a notification (e.g., vibrate the device), log the information to a content provider, post a message to a Web site, etc. Note that you will receive the `Intent` whenever the position is sampled and you are within the target zone – not just upon entering the zone. Hence, you will get the `Intent` several times, perhaps quite a few times depending on the size of the target zone and the speed of the device's movement.

Testing… Testing…

The Android emulator does not have the ability to get a fix from GPS, triangulate your position from cell towers, or identify your location by some nearby WiFi signal. So, if you want to simulate a moving device, you will need to have some means of providing mock location data to the emulator.

For whatever reason, this particular area has undergone significant changes as Android itself has evolved. It used to be that you could provide mock location data within your application, which was very handy for demonstration purposes. Alas, those options have all been removed as of Android 1.0.

One likely option for supplying mock location data is the Dalvik Debug Monitor Service (DDMS). This is an external program, separate from the emulator, where you can feed it single location points or full routes to traverse, in a few different formats, as is mentioned in the chapter on DDMS.

You can also send location fixes via `telnet` to an emulator. The port number is in your emulator's title bar (usually `5554` for the first running emulator instance). You can then run:

```
telnet localhost 5554
```

to access the Android Console within the emulator. Running the `geo fix NNN NNN` command, where `NNN NNN` is your latitude and longitude, will have the emulator respond as if those coordinates came from GPS.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Alternative Flavors of Updates

There are more ways to get updates from `LocationManager` than the versions of `requestLocationUpdates()` we have seen so far. There are four major axes of difference:

1. Some versions of `requestLocationUpdates()` take a `Criteria` object, having Android give you fixes based on the best-available provider given the requirements stipulated in the `Criteria`
2. Some versions of `requestLocationUpdates()` take a `Looper` as a parameter, allowing you to receive updates on a background `HandlerThread` instead of the main application thread
3. Some versions of `requestLocationUpdates()` take a `PendingIntent` which will be executed, instead of calling your `LocationListener`
4. There are a few flavors of `getSingleUpdate()`, which, as the name suggests, gives you just one location fix, rather than a stream until you remove the request for updates

For the `Criteria`-flavored versions of `requestLocationUpdates()` and `requestSingleUpdate()`, bear in mind that your code will still crash if there are no possible providers for your `Criteria`. For example, even if you use an empty `Criteria` object (for maximum possible matches), but GPS is disabled and the device lacks telephony (e.g., a tablet), you can get a crash like this one:

```
02-09 13:29:21.549: E/AndroidRuntime(2236): FATAL EXCEPTION: main
02-09 13:29:21.549: E/AndroidRuntime(2236): java.lang.RuntimeException: Unable
to resume activity {com.commonsware.android.mapsv2.location/
com.commonsware.android.mapsv2.location.MainActivity}:
java.lang.IllegalArgumentException: no providers found for criteria
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.app.ActivityThread.performResumeActivity(ActivityThread.java:2564)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.app.ActivityThread.handleResumeActivity(ActivityThread.java:2607)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2088)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.app.ActivityThread.access$600(ActivityThread.java:134)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.app.ActivityThread$H.handleMessage(ActivityThread.java:1233)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.os.Handler.dispatchMessage(Handler.java:99)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.os.Looper.loop(Looper.java:137)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.app.ActivityThread.main(ActivityThread.java:4699)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
java.lang.reflect.Method.invokeNative(Native Method)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
java.lang.reflect.Method.invoke(Method.java:511)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:787)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:554)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
dalvik.system.NativeStart.main(Native Method)
02-09 13:29:21.549: E/AndroidRuntime(2236): Caused by:
java.lang.IllegalArgumentException: no providers found for criteria
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.os.Parcel.readException(Parcel.java:1331)
02-09 13:29:21.549: E/AndroidRuntime(2236):   at
android.os.Parcel.readException(Parcel.java:1281)
02-09 13:29:21.549: E/AndroidRuntime(2236):   ... 19 more
```

Hence, you will still want to use `getProviders()` or `getBestProvider()` to ensure that your `Criteria` will resolve to *something* before you try using the `Criteria` to actually request fixes.

The Fused Option

Google Play Services — the proprietary API set supported by many Android devices – offers a fused location provider that simplifies location tracking. This capability is covered in the next chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Fused Location Provider

At the 2013 Google I|O conference, Google announced an update to Google Play Services that offers a "fused location provider", one that seamlessly uses all available location data to give you as accurate of a location as possible, as quickly as possible, with as little power consumption as possible. This serves as an adjunct to the traditional `LocationManager` approach for finding one's position. The fused location provider has a different API, though one that is similar in some respects to the `LocationManager` API.

In this chapter, we will examine how to use the fused location provider.

Prerequisites

This chapter assumes that you have read the preceding chapter on location-based services, along with that chapter's prerequisites.

Why Use the Fused Location Provider?

The traditional recipes for using location providers are a bit complicated, if you want to maximize results. Simply asking for a GPS fix is not that hard, but:

- What if GPS is disabled?
- What if GPS signals are unavailable (e.g., the device is indoors)?
- What about the GPS power drain?

The fused location provider is designed to address these sorts of concerns. Its implementation will blend data from GPS, cell tower triangulation, and WiFi hotspot proximity to determine the device's location, without your having to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

manually set all of that up. The fused location provider will also take advantage of sensor data, so it does not try to update your location as frequently if the accelerometer indicates that you are not moving.

The net result is better location data, delivered more quickly, with (reportedly) less power consumption.

Why Not Use the Fused Location Provider?

The fused location provider is part of Google Play Services. Google Play Services is available on hundreds of millions of Android devices. However:

- It is closed source, and so we do not know what the Play Services all do, and whether anything that it does might be detrimental.
- It is proprietary, and so Play Services will not be available on the Kindle Fire series and other devices working solely from the Android open source project.
- Play Services is only available on devices that have the Play Store, as opposed to the old Android Market, and so older devices (e.g., Android 2.2 and older) are far less likely to have Play Services available.

If you are aiming to distribute your app solely through the Play Store, relying upon the Play Services framework is reasonable. If, however, you are distributing through other channels, you will either need to *conditionally* use the fused location provider on devices that offer it, or *avoid* the fused location provider entirely, falling back to the traditional `LocationManager` solution.

Finding Our Location, Once

The fused location provider requires a fair bit of setup, because of its dependence upon the Play Services framework. However, once that is established, the fused location provider is as easy to use, if not easier, than is `LocationManager`.

This section will review the `Location/Fused` sample application, which is a clone of the `Internet/Weather` sample application from the previous chapter, revised to use the fused location provider to get a one-off weather forecast.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Installing and Attaching Google Play Services

If you have not done so already (e.g., for Maps V2), you will need to install the Play Services framework in your development environment.

Android Studio users should install the "Google Repository" entry in the SDK Manager. At that point, you can add a dependency upon the `com.google.android.gms:play-services-location` artifact for some appropriate version, such as `com.google.android.gms:play-services-location:6.5.87`.

Eclipse users will need to install the "Google Play services" entry in the SDK Manager, then add to your project a reference to the Android library project found in the `extras/google/google_play_services/libproject/google-play-services_lib/` directory in your Android SDK installation.

Checking for Google Play Services

At runtime, before we even start bothering to try the fused location provider, we need to confirm that Play Services is ready for use. As with the samples shown in the chapter on Maps V2, our revised `WeatherDemo` activity uses a `readyToGo()` method to confirm that it is safe to start the fragment:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  if (readyToGo()) {
    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new WeatherFragment()).commit();
    }
  }
}
```

In `readyToGo()`, we call the static `isGooglePlayServicesAvailable()` method on `GooglePlayServicesUtil`. This will return an integer indicating whether Play Services is available for our use or not:

```
protected boolean readyToGo() {
  int status=
      GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);

  if (status == ConnectionResult.SUCCESS) {
    return(true);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
    else if (GooglePlayServicesUtil.isUserRecoverableError(status)) {
      ErrorDialogFragment.newInstance(status)
                         .show(getFragmentManager(),
                               TAG_ERROR_DIALOG_FRAGMENT);
    }
    else {
      Toast.makeText(this, R.string.no_fused, Toast.LENGTH_LONG).show();
      finish();
    }

    return(false);
  }
```

If the return value is `ConnectionResult.SUCCESS`, we return `true`, to indicate that we are indeed ready to go. But, what if `isGooglePlayServicesAvailable()` returns something else?

There are two major possibilities here:

1. The error is something that the user might be able to rectify, such as by downloading the Google Play Services app from the Play Store
2. The error is something that the user cannot recover from

We can distinguish these two cases by calling the static `isUserRecoverableError()` on `GooglePlayServicesUtil`, passing in the value we received from `isGooglePlayServicesAvailable()`. This will return `true` if the user might be able to fix the problem, `false` otherwise.

In the `false` case, the user is just out of luck, so we display a `Toast` to alert them of this fact, then `finish()` the activity and return `false`, so `WeatherDemo` skips over the rest of its work.

In the `true` case, we can display something to the user to prompt them to fix the problem. One way to do that is to use a dialog obtained from Google code, by calling the static `getErrorDialog()` method on the `GooglePlayServicesUtil` class. In our case, we wrap that in a `DialogFragment` named `ErrorDialogFragment`, implemented as a static inner class of `AbstractMapActivity`:

```
  public static class ErrorDialogFragment extends DialogFragment {
    static final String ARG_STATUS="status";

    static ErrorDialogFragment newInstance(int status) {
      Bundle args=new Bundle();

      args.putInt(ARG_STATUS, status);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    ErrorDialogFragment result=new ErrorDialogFragment();

    result.setArguments(args);

    return(result);
  }

  @Override
  public Dialog onCreateDialog(Bundle savedInstanceState) {
    Bundle args=getArguments();

    return GooglePlayServicesUtil.getErrorDialog(args.getInt(ARG_STATUS),
                                                 getActivity(), 0);
  }

  @Override
  public void onDismiss(DialogInterface dlg) {
    if (getActivity() != null) {
      getActivity().finish();
    }
  }
}
```

While the code and comments around `getErrorDialog()` suggest that there is some way for us to find out if the user performed actions that fix the problem, this code does not seem to work well in practice. After all, downloading Google Play Services is asynchronous, so even if the user returns to our app, it is entirely likely that Play Services is still unavailable. As a result, when the user is done with the dialog, we `finish()` the activity, forcing the user to start it again if and when they are done downloading Play Services.

As it turns out, not all Android devices support the Play Store, or the Google Play Services by extension. This leads to unpleasant user experiences:

- If the device lacks the Play Store, `isUserRecoverableError()` returns `true`, even though the user cannot recover from this situation (except perhaps via a firmware update)
- `getErrorDialog()` apparently can return `null` for cases where the error is supposedly user-recoverable

Permissions

To use the fused location provider, you still need the `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION` permissions. If you only hold `ACCESS_COARSE_LOCATION`, the data you get back will be limited to data that is sufficiently “fuzzy”. Typically, if

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

you are bothering using this provider, you will request `ACCESS_FINE_LOCATION` — if coarse location data is all you need, using `LocationManager` should be just as good and is compatible with more devices.

Clients, Connections, and Callbacks

Play Services runs in its own process, one that appears to be continuously monitoring the user's location. In order to get location data from this process, we need to establish some sort of IPC (inter-process communication) with it. The low-level implementation of this is handled by the Play Services Android library project. However, we do need to set some things up ourselves.

Specifically, we need to create and use an instance of `LocationClient`, our gateway to the Play Services fused location provider. In the sample app, we create this `GoogleApiClient` in `onCreate()` of our `WeatherFragment`, holding onto the object in a `client` data member:

```
public class WeatherFragment extends ListFragment implements
    GoogleApiClient.ConnectionCallbacks,
    GoogleApiClient.OnConnectionFailedListener,
    Runnable {
  private static final int REQUEST_ID=1337;
  private String template=null;
  private GoogleApiClient client=null;
```

We initialize the `GoogleApiClient` in `onAttach()`, as we now know that we have an activity to work with:

```
  @Override
  public void onAttach(Activity host) {
    super.onAttach(host);

    client=new GoogleApiClient.Builder(host)
                .addApi(LocationServices.API)
                .addConnectionCallbacks(this)
                .addOnConnectionFailedListener(this)
                .build();
  }
```

To create a `GoogleApiClient` instance, use a `GoogleApiClient.Builder`. As the class name suggests, `GoogleApiClient` is used as a client connection to many (but not all) Google Play Services APIs, and `GoogleApiClient.Builder` is a builder for building such a connection. In particular:

- We pass our `Activity` as our `Context`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- We call `addApi(LocationServices.API)` to indicate that we are interested in the fused locations API
- We call `addConnectionCallbacks()` to indicate that the fragment should be the one to be notified when our connection to the Play Services process is ready
- We call `addOnConnectionFailedListener()` to indicate that the fragment should be the one to be notified if we have a problem connecting to the Play Services process
- We call `build()` on the `Builder` to actually build the `GoogleApiClient`

Given that we have a `GoogleApiClient`, we need to `connect()` to it to be able to start requesting location data, then `disconnect()` from it when we no longer need that location data. In our case, we do that work in `onResume()` and `onPause()` of our `WeatherFragment`:

```
@Override
public void onResume() {
  super.onResume();

  client.connect();
}

@Override
public void onPause() {
  getListView().removeCallbacks(this);
  client.disconnect();

  super.onPause();
}
```

(the call to `removeCallbacks()` will be explained shortly)

These, in turn, will trigger calls to our `onConnected()` and `onDisconnected()` methods of the `GoogleApiClient.ConnectionCallbacks` interface, assuming all goes well. In `onConnected()`, we can start requesting location data, which we delegate to the `run()` method of the `WeatherFragment` (implemented as part of implementing the `Runnable` interface). Our fragment largely ignores `onConnectionSuspended()`, but if we needed to stop something else once we no longer were eligible for location data, we could stop that here.

```
@Override
public void onConnected(Bundle undocumented) {
  run();
}

@Override
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public void onConnectionSuspended(int i) {
  Log.w(((Object)this).getClass().getSimpleName(),
        "onConnectionSuspended() called, whatever that means");
}
```

However, apparently it is possible for this connection attempt to fail. Exactly how and why it might fail is not well documented. If it fails, the `onConnectionFailed()` method from our `GoogleApiClient.OnConnectionFailedListener` implementation will be called. `onConnectionFailed()` is passed a `ConnectionResult` indicating what specifically went wrong.

It turns out that this `ConnectionResult` may contain a `PendingIntent` that can be used to try to help the user recover from whatever the problem was. The recipe that we have been given to try to use this is to call `hasResolution()` (to see if the `PendingIntent` exists) and to use `startResolutionForResult()` (to invoke the activity pointed to by the `PendingIntent`). Of course, `hasResolution()` may return false, and apparently the `PendingIntent` might be broken, so we have to handle those scenarios as well:

```
@Override
public void onConnectionFailed(ConnectionResult result) {
  boolean anyLuck=false;

  if (result.hasResolution()) {
    try {
      result.startResolutionForResult(getActivity(), REQUEST_ID);
      anyLuck=true;
    }
    catch (IntentSender.SendIntentException e) {
      Log.e(((Object)this).getClass().getSimpleName(),
            "Exception trying to startResolutionForResult()", e);
    }
  }

  if (!anyLuck) {
    Toast.makeText(getActivity(), R.string.no_fused,
                   Toast.LENGTH_LONG).show();
    getActivity().finish();
  }
}
```

If we have a resolution and successfully start up the resolution activity, our fragment will be paused and later resumed, at which point we will wind up trying to `connect()` again naturally. If we do not have a resolution, or the resolution itself is broken, the sample just displays a `Toast` and finishes the activity, along the same lines as how we handle the case where Play Services is completely unavailable. A production-grade application would probably handle this case a bit more gracefully.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Finding the Current Location

Given all that setup, actually getting the location is almost anti-climactic.

To find the current location, given a connected `GoogleApiClient`, just call the static `getLastLocation()` method on `LocationServices.FusedLocationApi`, passing in the `GoogleApiClient` instance as a parameter. This usually will return a non-`null` `Location` object, using the same `Location` class that you would use with `LocationManager`.

In the sample, the `run()` method checks to see if `getLastLocation()` returns `null` or not. If the location is `null`, it schedules `run()` to be invoked again in one second, using `postDelayed()` on some suitable `View` (in this case, the `WebView` for displaying the results). If, however, we do have a valid location, `run()` invokes a `FetchForecastTask`, as did the original version of this sample:

```
@Override
public void run() {
  Location loc=LocationServices.FusedLocationApi
                  .getLastLocation(client);

  if (loc == null) {
    getListView().postDelayed(this, 1000);
  }
  else {
    FetchForecastTask task=new FetchForecastTask();

    task.execute(loc);
  }
}
```

The fact that we are using `postDelayed()` here is why we use `removeCallbacks()` in `onPause()`, to stop polling for `getLastLocation()` when we are disconnecting from the `LocationClient`.

Note that the documentation for `getLastLocation()` states "If a location is not available, which should happen very rarely, `null` will be returned." The "very rarely" part indicates that Play Services is constantly checking for the user's location, if it is able to do so.

The Rest of the Sample

The rest of the sample follows some examples from earlier in the book for fetching data and displaying it.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`run()`, once it gets a location from `getLastLocation()`, executes a `FetchForecastTask`, which is a subclass of `AsyncTask`. This task, in `doInBackground()`, calls `getForecastXML()`, which uses `HttpUrlConnection` to retrieve the weather forecast XML. The task also calls `buildForecasts()`, which parses that XML using the Java DOM and builds an `ArrayList` of `Forecast` objects, where each `Forecast` holds a time, the predicted temperature, and a code indicating the expected cloud cover and precipitation.

In the task's `onPostExecute()` method, it creates a `ForecastAdapter`, wrapped around our list of `Forecast` objects, and puts that adapter in the `ListView` for our `ListFragment`. `ForecastAdapter` follows the same pattern as was seen in the Picasso sample, using Picasso to load in icons associated with each cloud cover/precipitation prediction and show those alongside the time and temperature.

The result looks like:

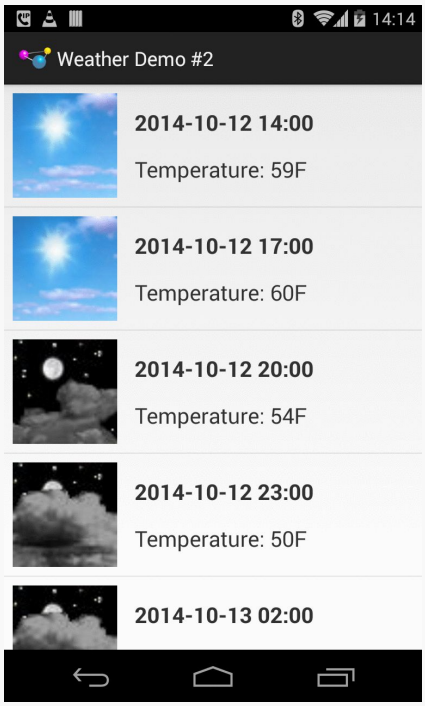


Figure 667: The FusedNew Sample Application

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Requesting Location Updates

As with `LocationManager`, you can use `LocationClient` to be delivered location updates as the device moves, via `requestLocationUpdates()`. There are two major axes of control you have over these updates: the way the locations are delivered to you, and the `LocationRequest` that configures what updates you receive.

Delivery Options

A foreground application would use forms of `requestLocationUpdates()` that take a `LocationListener` as a parameter. Despite the class being named the same, this is a separate implementation of the `LocationListener` interface. The Play Services one (`com.google.android.gms.location.LocationListener`) only requires a single method: `onLocationChanged()`, which is handed a `Location` object representing a location fix.

A background application would use the `requestLocationUpdates()` that takes a `PendingIntent` instead of a `LocationListener`, where that `PendingIntent` can do whatever you wish (start an activity, start a service, send a broadcast). The location itself is delivered in the form of an `Intent` extra, keyed as `KEY_LOCATION_CHANGED`, with a value in the form of a `Location` object.

Request Options

All forms of `requestLocationUpdates()` take a `LocationRequest` object describing what you want in terms of updates. Unlike with `LocationManager`, you do not specify specific location technologies (e.g., GPS). You also lack the fine-grained control of the `Criteria` object (e.g., to require the location to have speed data). However, you do have some measure of control, via various setters on `LocationRequest`.

Frequency

Calling `setInterval()` indicates approximately how frequently you wish to receive location updates. The key word here is "approximately", as you will receive updates more or less frequently than the number of milliseconds you specify as the desired interval. However, your requested interval is taken into account, and the longer of an interval you provide, the less power your app will consume.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To help prevent being flooded with location data, you can also call `setFastestInterval()`, which will throttle the actual updates to be no more frequent than the number of milliseconds that you state.

Priority

`setPriority()` allows you to control the accuracy and power consumption of your app's request, by specifying one of four possible priority levels:

- `PRIORITY_HIGH_ACCURACY` will tend to use GPS and therefore will consume more power
- `PRIORITY_BALANCED_POWER_ACCURACY` will try to consume somewhat less power
- `PRIORITY_LOW_POWER` will try to consume even less power
- `PRIORITY_NO_POWER` indicates that you want to consume no additional power over any other requests, but to get what you can (akin to the "passive provider" available with `LocationManager`)

Duration

You can proactively cancel receiving further updates by calling `removeUpdates()`, passing in your delivery option from `requestLocationUpdates()`:

- The same `LocationListener` as you used to request the updates, or
- An equivalent `PendingIntent` to the one that you used to request the updates

You can also automatically expire your requested updates by one of three means:

- `setNumUpdates()` indicates exactly how many location fixes that you want to receive (e.g., 1) and discontinues the updates after that number
- `setExpirationDuration()` indicates how long you wish to receive updates, expressed as a number of milliseconds from now
- `setExpirationTime()` indicates when you wish to discontinue updates, expressed in the form of the number of milliseconds since the device turned on (e.g., the same time base as is used by `elapsedRealtime()` on the `SystemClock` class)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For example, an improved version of the sample shown in this chapter would use a `LocationRequest` with `setNumUpdates(1)`, instead of the one-second polling of `getLastLocation()`.

Gaps in the Fused Location Provider

The fused location provider is governed by the user's enabled location providers (e.g., via the Settings app). Unfortunately, there is no way to interrogate `LocationClient` to find out whether it is even possible for you to get a location, such as if all providers are disabled. You just fail to get location data (e.g., `getLastLocation()` perpetually returns `null`). If you need to detect this case, you will also need to use `LocationManager` and methods like `isProviderEnabled()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Working with the Clipboard

Being able to copy and paste is something that mobile device users seem to want almost as much as their desktop brethren. Most of the time, we think of this as copying and pasting text, and for a long time that was all that was possible on Android. Android 3.0 added in new clipboard capabilities for more rich content, which application developers can choose to support as well. This section will cover both of these techniques.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Using the Clipboard on Android 1.x/2.x

Android has a `ClipboardManager` that allows you to interact with the clipboard manually, in addition to built-in clipboard facilities for users (e.g., copy/paste context menus on `EditText`). `ClipboardManager`, like `AudioManager`, is obtained via a call to `getSystemService()`:

From there, you have three simple methods:

1. `getText()` to retrieve the current clipboard contents
2. `hasText()`, to determine if there are any clipboard contents, so you can react accordingly (e.g., disable "paste" menus when there is nothing to paste)
3. `setText()`, to put text on the clipboard

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For example, the SystemServices/ClipIP sample project contains a little application that puts your current IP address on the clipboard, for pasting into some EditText of an application.

The IPClipper activity's onCreate() does the work of putting text onto the clipboard via setText() and notifying the user via a Toast:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  try {
    String addr=getLocalIPAddress();

    if (addr == null) {
      Toast.makeText(this,
                     "IP address not available -- are you online?",
                     Toast.LENGTH_LONG).show();
    }
    else {
      ClipboardManager cm=
          (ClipboardManager)getSystemService(CLIPBOARD_SERVICE);

      try {
        cm.setText(addr);

        Toast.makeText(this, "IP Address clipped!",
                       Toast.LENGTH_SHORT).show();
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(), "Exception clipping IP", e);
        Toast.makeText(this, "Exception: " + e.getMessage(),
                       Toast.LENGTH_SHORT).show();
      }
    }
  }
  catch (Exception e) {
    Log.e("IPClipper", "Exception getting IP address", e);
    Toast.makeText(this, "Could not obtain IP address",
                   Toast.LENGTH_LONG).show();
  }

  finish();
}
```

The work of figuring out what the IP address is can be found in the getLocalIPAddress() method:

```
public String getLocalIPAddress() throws SocketException {
  Enumeration<NetworkInterface> nics=
      NetworkInterface.getNetworkInterfaces();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    while (nics.hasMoreElements()) {
      NetworkInterface intf=nics.nextElement();
      Enumeration<InetAddress> addrs=intf.getInetAddresses();

      while (addrs.hasMoreElements()) {
        InetAddress addr=addrs.nextElement();

        if (!addr.isLoopbackAddress()) {
          return(addr.getHostAddress().toString());
        }
      }
    }

    return(null);
  }
```

This uses the `NetworkInterface` and `InetAddress` classes from the `java.net` package to loop through all network interfaces and find the first one that has a non-localhost (loopback) IP address. The emulator will return 10.0.2.15 all of the time; your device will return whatever IP address it has from WiFi, 3G, etc. If no such address is available, it returns `null`.

The activity itself has no UI, as the application uses `Theme.NoDisplay`. The activity avoids a call to `setContentView()`, and calls `finish()` when all the work is done in `onCreate()`. Hence after starting the activity, the user will hopefully see the "successful" `Toast`, and nothing else:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 668: The IPClipper, shortly after launching

Then, if the user long-taps on an `EditText` and chooses Paste, the IP address is added to the `EditText` contents. Note that the clipboard is system-wide, not merely application-wide, which is why you can successfully paste the IP address into any application's fields.

Note that there is a significant bug in Android 4.3 that, until it is fixed, will require you to do a bit more error-handling with your clipboard operations. That is why we have our `setText()` call wrapped in a `try/catch` blog, even though `setText()` does not throw a checked exception. The rationale for this will be discussed later in this chapter.

Advanced Clipboard on Android 3.x and Higher

Android 3.0 added in new ways of working with `ClipboardManager` to clip things that transcend simple text. In part, this is expected to be used for advanced copy and paste features between applications. However, this also forms the foundation for a rich drag-and-drop model within an application.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that they also moved `ClipboardManager` to the `android.content` package. You can still refer to it via the `android.text` package, for backwards compatibility. However, if your project will be on API Level 11 or higher only, you might consider using the new `android.content` package edition of the class.

Copying Rich Data to the Clipboard

In addition to methods like `setText()` to put a piece of plain text on the clipboard, `ClipboardManager` (as of API Level 11) offers `setPrimaryClip()`, which allows you to put a `ClipData` object on the clipboard.

What's a `ClipData`? In some respects, it is whatever you want. It can hold:

1. plain text
2. a `Uri` (e.g., to a piece of music)
3. an `Intent`

The `Uri` means that you can put anything on the clipboard that can be referenced by a `Uri`... and if there is nothing in Android that lets you reference some data via a `Uri`, you can invent your own content provider to handle that chore for you. Furthermore, a single `ClipData` can actually hold as many of these as you want, each represented as individual `ClipData.Item` objects. As such, the possibilities are endless.

There are static factory methods on `ClipData`, such as `newUri()`, that you can use to create your `ClipData` objects. In fact, that is what we use in the SystemServices/ClipMusic sample project and the `MusicClipper` activity.

`MusicClipper` has the classic two-big-button layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <Button android:id="@+id/pick"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="Pick"
    android:onClick="pickMusic"
  />
  <Button android:id="@+id/view"
    android:layout_width="match_parent"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="Play"
    android:onClick="playMusic"
  />
</LinearLayout>
```

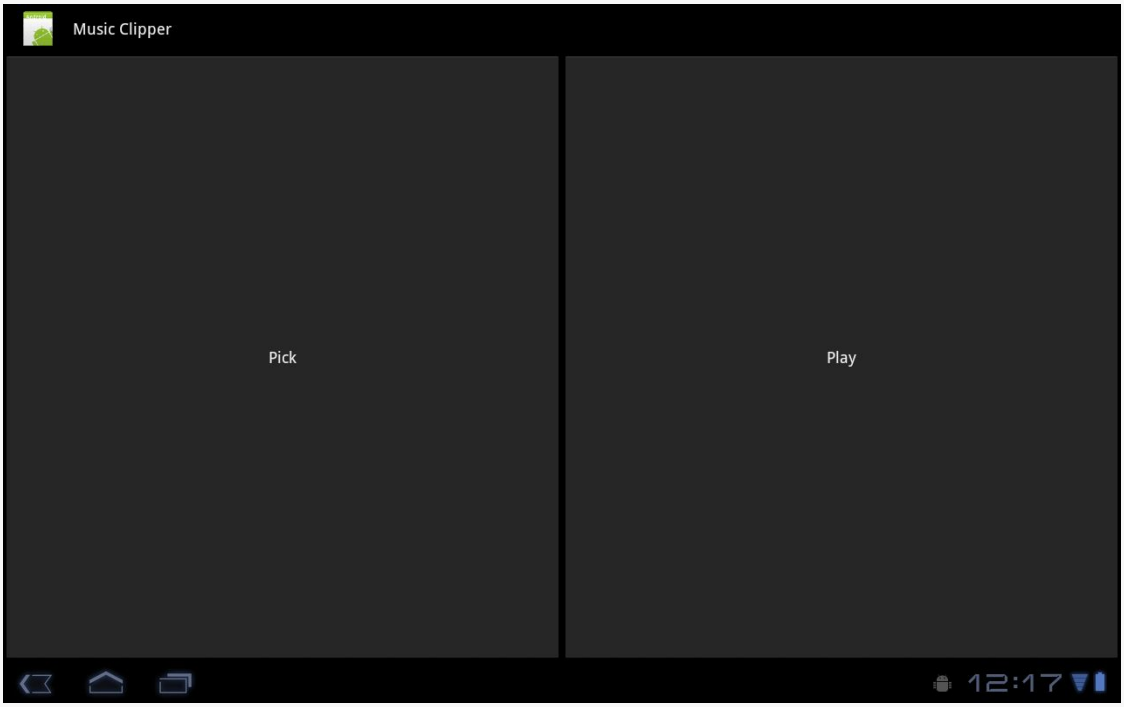


Figure 669: The Music Clipper main screen

In `onCreate()`, we get our hands on our `ClipboardManager` system service:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    clipboard=(ClipboardManager)getSystemService(CLIPBOARD_SERVICE);
  }
```

Tapping the "Pick" button will let you pick a piece of music, courtesy of the `pickMusic()` method wired to that `Button` object:

```
  public void pickMusic(View v) {
    Intent i=new Intent(Intent.ACTION_GET_CONTENT);

    i.setType("audio/*");
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    startActivityForResult(i, PICK_REQUEST);
  }
```

Here, we tell Android to let us pick a piece of music from any available audio MIME type (`audio/*`). Fortunately, Android has an activity that lets us do that:

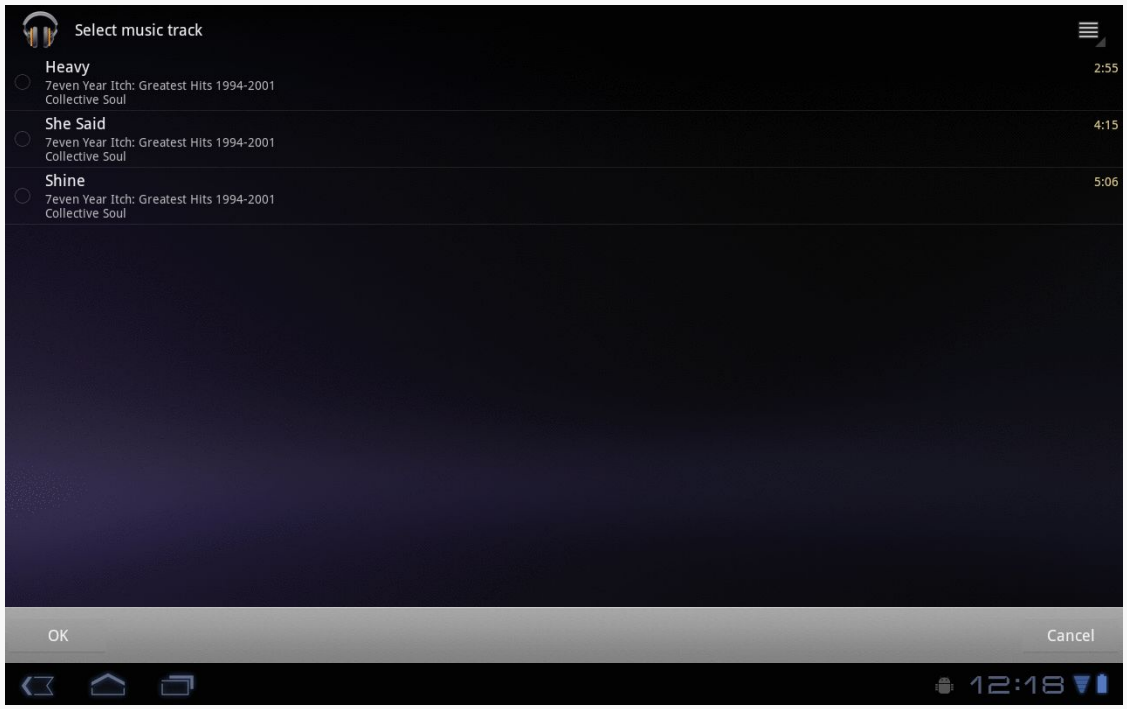


Figure 670: The XOOM tablet's music track picker

We get the result in `onActivityResult()`, since we used `startActivityForResult()` to pick the music. There, we package up the `content://` `Uri` to the music into a `ClipData` object and put it on the clipboard:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (requestCode == PICK_REQUEST) {
    if (resultCode == RESULT_OK) {
      ClipData clip=
          ClipData.newUri(getContentResolver(), "Some music",
                          data.getData());

      try {
        clipboard.setPrimaryClip(clip);
      }
      catch (Exception e) {
        Log.e(getClass().getSimpleName(), "Exception clipping Uri", e);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
          Toast.makeText(this, "Exception: " + e.getMessage(),
                         Toast.LENGTH_SHORT).show();
        }
      }
    }
  }
```

Note that there is a significant bug in Android 4.3 that, until it is fixed, will require you to do a bit more error-handling with your clipboard operations. That is why we have our `setPrimaryClip()` call wrapped in a `try/catch` blog, even though `setPrimaryClip()` does not throw a checked exception. The rationale for this will be discussed later in this chapter.

Pasting Rich Data from the Clipboard

The catch with rich data on the clipboard is that somebody has to know about the sort of information you are placing on the clipboard. Eventually, the Android development community will work out common practices in this area. Right now, though, you can certainly use it within your own application (e.g., clipping a note and pasting it into another folder).

Since putting `ClipData` onto the clipboard involves a call to `setPrimaryClip()`, it should not be surprising that the reverse operation — getting a `ClipData` from the clipboard — uses `getPrimaryClip()`. However, since you do not know where this clip came from, you need to validate that it has what you expect and to let the user know when the clipboard contents are not something you can leverage.

The "Play" button in our UI is wired to a `playMusic()` method. This will only work when we have pasted a `Uri ClipData` to the clipboard pointing to a piece of music. Since we cannot be sure that the user has done that, we have to sniff around:

```
public void playMusic(View v) {
  ClipData clip=clipboard.getPrimaryClip();

  if (clip == null) {
    Toast.makeText(this, "There is no clip!", Toast.LENGTH_LONG)
         .show();
  }
  else {
    ClipData.Item item=clip.getItemAt(0);
    Uri song=item.getUri();

    if (song != null
        && getContentResolver().getType(song).startsWith("audio/")) {
      startActivity(new Intent(Intent.ACTION_VIEW, song));
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      else {
        Toast.makeText(this, "There is no song!", Toast.LENGTH_LONG)
             .show();
      }
    }
  }
```

First, there may be nothing on the clipboard, in which case the `ClipData` returned by `getPrimaryClip()` would be `null`. Or, there may be stuff on the clipboard, but it may not have a `Uri` associated with it (`getUri()` on `ClipData`). Even then, the `Uri` may point to something other than music, so even if we get a `Uri`, we need to use a `ContentResolver` to check the MIME type (`getContentResolver().getType()`) and make sure it seems like it is music (e.g., starts with `audio/`). Then, and only then, does it make sense to try to start an `ACTION_VIEW` activity on that `Uri` and hope that something useful happens. Assuming you clipped a piece of music with the "Pick" button, "Play" will kick off playback of that song.

ClipData and Drag-and-Drop

Android 3.0 also introduced Android's first built-in drag-and-drop framework. One might expect that this would be related entirely to `View` and `ViewGroup` objects and have nothing to do with the clipboard. In reality, the drag-and-drop framework leverages `ClipData` to say what it is that is being dragged and dropped. You call `startDrag()` on a `View`, supplying a `ClipData` object, along with some objects to help render the "shadow" that is the visual representation of this drag operation. A `View` that can receive objects "dropped" via drag-and-drop needs to register an `OnDragListener` to receive drag events as the user slides the shadow over the top of the `View` in question. If the user lifts their finger, thereby dropping the shadow, the recipient `View` will get an `ACTION_DROP` drag event, and can get the `ClipData` out of the event.

Monitoring the Clipboard

Android 3.0 added the capability for an app to monitor what is put on the clipboard, including things put on the clipboard by other apps.

This is a somewhat esoteric feature, but one that perhaps has some valid use cases. Mostly, it would be used by something *not* in the foreground, since the foreground activity is probably what is adding material to the clipboard. A service, or perhaps an activity that has moved to the background, could use this feature to find out about new clipboard entries.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To monitor the clipboard, you simply call `addPrimaryClipChangedListener()` on `ClipboardMonitor`, passing an implementation of an `OnPrimaryClipChangedListener` interface. That object, in turn, will be called with `onPrimaryClipChanged()` whenever there is a new clipboard entry. Later on, you can call `removePrimaryClipChangedListener()` to stop being notified about new clipboard entries.

For example, here is `MainActivity` from the SystemServices/ClipboardMonitor sample project:

```
package com.commonsware.android.clipmon;

import android.app.Activity;
import android.content.ClipboardManager;
import android.content.ClipboardManager.OnPrimaryClipChangedListener;
import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends Activity implements
    OnPrimaryClipChangedListener {
  private ClipboardManager cm=null;
  private TextView lastClip=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    lastClip=(TextView)findViewById(R.id.last_clip);
    cm=(ClipboardManager)getSystemService(CLIPBOARD_SERVICE);
  }

  @Override
  public void onResume() {
    super.onResume();
    cm.addPrimaryClipChangedListener(this);
  }

  @Override
  public void onPause() {
    cm.removePrimaryClipChangedListener(this);
    super.onPause();
  }

  @Override
  public void onPrimaryClipChanged() {
    lastClip.setText(cm.getPrimaryClip().getItemAt(0)
                       .coerceToText(this));
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here, we:

- Retrieve the `ClipboardManager` in `onCreate()`
- Register for clipboard events via `addPrimaryClipChangedListener()` in `onResume()`
- Unregister from clipboard events via `removePrimaryClipChangedListener()` in `onPause()`
- Convert the first item (`getItemAt(0)`) of the primary clip (`getPrimaryClip()`) to text (`coerceToText(this)`), and stuff the results into a `TextView`

In theory, this activity will display new clipboard entries as they arrive. In practice, it will only do so while it is in the foreground, and so it would require something in the background to add something to the clipboard. That is not a particularly useful example... except to test the bug outlined in the next section.

The Android 4.3 Clipboard Bug

AndroidPolice reported on a fairly unpleasant bug in Android 4.3. While this bug was fixed in Android 4.4, there is little evidence that Google will be releasing a fix for Android 4.3 devices, which means that this problem will plague developers into 2015 and perhaps beyond.

The bug stems from the clipboard monitoring facility. If an app has used `addPrimaryClipChangedListener()`, *any other app* that tries to paste to the clipboard will crash.

The first crash will be a `SecurityException`:

```
java.lang.SecurityException: uid ... does not have
android.permission.UPDATE_APP_OPS_STATS
```

The second and subsequent times this occurs on the device, it will be an `IllegalStateException`:

```
java.lang.IllegalStateException: beginBroadcast() called while already
in a broadcast
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The only resolution is to unregister the clipboard listener... and hope that the first crash has not occurred. If it has, a **full reboot of the device is required** to fix the broken system.

If Your App Monitors the Clipboard…

If you have a component, such as a long-running service, that is monitoring the clipboard, please ensure that the users have an easy way to stop that behavior, even if it means stopping your whole service. While this may mean that your app has seriously degraded functionality, the alternative is that the user has to keep rebooting their device while your app is installed.

If Your App Pastes to the Clipboard…

If you are pasting to the clipboard, with `setPrimaryClip()` or the older `setText()`, you will want to throw a `try/catch` block around those calls, so you catch the `RuntimeExceptions` that will be thrown.

However, you will need to tell your users that they are now fairly well screwed, needing to both find the clipboard-monitoring app and learn how to control it (or uninstall/disable it, if needed), plus reboot their device, in order to paste to the clipboard again.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Telephony

Many, if not most, Android devices will be phones. As such, not only will users be expecting to place and receive calls using Android, but you will have the opportunity to help them place calls, if you wish.

Why might you want to?

1. Maybe you are writing an Android interface to a sales management application (a la Salesforce.com) and you want to offer users the ability to call prospects with a single button click, and without them having to keep those contacts both in your application and in the phone's contacts application
2. Maybe you are writing a social networking application, and the roster of phone numbers that you can access shifts constantly, so rather than try to "sync" the social network contacts with the phone's contact database, you let people place calls directly from your application
3. Maybe you are creating an alternative interface to the existing contacts system, perhaps for users with reduced motor control (e.g., the elderly), sporting big buttons and the like to make it easier for them to place calls

Whatever the reason, Android has the means to let you manipulate the phone just like any other piece of the Android system.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on working with multiple activities.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Report To The Manager

To get at much of the phone API, you use the `TelephonyManager`. That class lets you do things like:

1. Determine if the phone is in use via `getCallState()`, with return values of `CALL_STATE_IDLE` (phone not in use), `CALL_STATE_RINGING` (call requested but still being connected), and `CALL_STATE_OFFHOOK` (call in progress)
2. Find out the SIM ID (IMSI) via `getSubscriberId()`
3. Find out the phone type (e.g., GSM) via `getPhoneType()` or find out the data connection type (e.g., GPRS, EDGE) via `getNetworkType()`

You Make the Call!

You can also initiate a call from your application, such as from a phone number you obtained through your own Web service. To do this, simply craft an `ACTION_DIAL` `Intent` with a `Uri` of the form `tel:NNNNN` (where `NNNNN` is the phone number to dial) and use that `Intent` with `startActivity()`. This will not actually dial the phone; rather, it activates the dialer activity, from which the user can then press a button to place the call.

For example, let's look at the Phone/Dialer sample application. Here's the crude-but-effective layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    >
    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Number to dial:"
      />
    <EditText android:id="@+id/number"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:cursorVisible="true"
      android:editable="true"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      android:singleLine="true"
    />
  </LinearLayout>
  <Button android:id="@+id/dial"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text="Dial It!"
    android:onClick="dial"
  />
</LinearLayout>
```

We have a labeled field for typing in a phone number, plus a button for dialing said number.

The Java code simply launches the dialer using the phone number from the field:

```
package com.commonsware.android.dialer;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class DialerDemo extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
  }

  public void dial(View v) {
    EditText number=(EditText)findViewById(R.id.number);
    String toDial="tel:"+number.getText().toString();

    startActivity(new Intent(Intent.ACTION_DIAL, Uri.parse(toDial)));
  }
}
```

The activity's own UI is not that impressive:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 671: The DialerDemo sample application, as initially launched

However, the dialer you get from clicking the dial button is better, showing you the number you are about to dial:

Figure 672: The Android Dialer activity, as launched from DialerDemo

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

No, Really, You Make the Call!

The good news is that `ACTION_DIAL` works without any special permissions. The bad news is that it only takes the user to the Dialer – the user still has to take action (pressing the green call button) to actually place the phone call.

An alternative approach is to use `ACTION_CALL` instead of `ACTION_DIAL`. Calling `startActivity()` on an `ACTION_CALL` `Intent` will immediately place the phone call, without any other UI steps required. However, you need the `CALL_PHONE` permission in order to use `ACTION_CALL`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Working With SMS

Oh, what a tangled web we weave

When first we practice to work with SMS on Android, Eve

(with apologies to Sir Walter Scott)

Android devices have had SMS capability since Android 1.0. However, from a programming standpoint, for years, SMS and Android were intensely frustrating. When the Android SDK was developed, some aspects of working with SMS were put into the SDK, while others were held back. This, of course, did not stop many an intrepid developer from working with the undocumented, unsupported SMS APIs, with varying degrees of success.

After much wailing and gnashing of teeth by developers, Google finally formalized a more complete SMS API in Android 4.4. However, this too has its issues, where some apps that worked fine with the undocumented API will now fail outright, in irreparable fashion, on Android 4.4+.

This chapter starts with the one thing you can do reasonably reliably across Android device versions – send an SMS, either directly or by invoking the user's choice of SMS client. The chapter then examines how to monitor or receive SMS messages (both pre–4.4 and 4.4+) and the SMS-related `ContentProvider` (both pre–4.4 and 4.4+).

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapters on broadcast `Intents`. One of the samples uses the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

ContactsContract provider, so reading that chapter will help you understand that particular sample.

Sending Out an SOS, Give or Take a Letter

While much of Android's SMS capabilities are not in the SDK, sending an SMS is. You have two major choices for doing this:

- Invoke the user's choice of SMS client application, so they can compose a message, track its progress, and so forth using that tool
- Send the SMS directly yourself, bypassing any existing client

Which of these is best for you depends on what your desired user experience is. If you are composing the message totally within your application, you may want to just send it. However, as we will see, that comes at a price: an extra permission.

Sending Via the SMS Client

Sending an SMS via the user's choice of SMS client is very similar to the use of ACTION_SEND described elsewhere in this book. You craft an appropriate Intent, then call startActivity() on that Intent to bring up an SMS client (or allow the user to choose between clients).

The Intent differs a bit from the ACTION_SEND example:

1. You use ACTION_SENDTO, rather than ACTION_SEND
2. Your Uri needs to begin with smsto:, followed by the mobile number you want to send the message to
3. Your text message goes in an sms_body extra on the Intent

For example, here is a snippet of code from the SMS/Sender sample project:

```
      Intent sms=new Intent(Intent.ACTION_SENDTO,
                            Uri.parse("smsto:"+c.getString(2)));

      sms.putExtra("sms_body", msg.getText().toString());

      startActivity(sms);
```

Here, our phone number is coming out of the third column of a Cursor, and the text message is coming from an EditText — more on how this works later in this section, when we review the Sender sample more closely.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Sending SMS Directly

If you wish to bypass the UI and send an SMS directly, you can do so through the `SmsManager` class, in the `android.telephony` package. Unlike most Android classes ending in `Manager`, you obtain an `SmsManager` via a static `getDefault()` method on the `SmsManager` class. You can then call `sendTextMessage()`, supplying:

1. The phone number to send the text message to
2. The "service center" address — leave this `null` unless you know what you are doing
3. The actual text message
4. A pair of `PendingIntent` objects to be executed when the SMS has been sent and delivered, respectively

If you are concerned that your message may be too long, use `divideMessage()` on `SmsManager` to take your message and split it into individual pieces. Then, you can use `sendMultipartTextMessage()` to send the entire `ArrayList` of message pieces.

For this to work, your application needs to hold the `SEND_SMS` permission, via a child element of your `<manifest>` element in your `AndroidManifest.xml` file.

For example, here is code from `Sender` that uses `SmsManager` to send the same message that the previous section sent via the user's choice of SMS client:

```
      SmsManager
        .getDefault()
        .sendTextMessage(c.getString(2), null,
                         msg.getText().toString(),
                         null, null);
```

Inside the Sender Sample

The `Sender` example application is fairly straightforward, given the aforementioned techniques.

The manifest has both the `SEND_SMS` and `READ_CONTACTS` permissions, because we want to allow the user to pick a mobile phone number from their list of contacts, rather than type one in by hand:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.sms.sender"
  android:installLocation="preferExternal"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission android:name="android.permission.READ_CONTACTS"/>
  <uses-permission android:name="android.permission.SEND_SMS"/>

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="Sender"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

If you noticed the `android:installLocation` attribute in the root element, that is to allow this application to be installed onto external storage, such as an SD card.

The layout has a `Spinner` (for a drop-down of available mobile phone numbers), a pair of `RadioButton` widgets (to indicate which way to send the message), an `EditText` (for the text message), and a "Send" `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
>
  <Spinner android:id="@+id/spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <RadioGroup android:id="@+id/means"
    android:orientation="horizontal"
    android:layout_width="match_parent"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:layout_height="wrap_content"
    >
      <RadioButton android:id="@+id/client"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:text="Via Client" />
      <RadioButton android:id="@+id/direct"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Direct" />
  </RadioGroup>
  <EditText
    android:id="@+id/msg"
    android:layout_width="match_parent"
    android:layout_height="0px"
    android:layout_weight="1"
    android:singleLine="false"
    android:gravity="top|left"
  />
  <Button
    android:id="@+id/send"
    android:text="Send!"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="sendTheMessage"
  />
</LinearLayout>
```

`Sender` uses the same technique for obtaining mobile phone numbers from our contacts as is seen in the chapter on contacts. To support Android 1.x and Android 2.x devices, we implement an abstract class and two concrete implementations, one for the old API and one for the new. The abstract class then has a static method to get at an instance suitable for the device the code is running on:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.os.Build;
import android.widget.SpinnerAdapter;

abstract class ContactsAdapterBridge {
  abstract SpinnerAdapter buildPhonesAdapter(Activity a);

  public static final ContactsAdapterBridge INSTANCE=buildBridge();

  private static ContactsAdapterBridge buildBridge() {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk<5) {
      return(new OldContactsAdapterBridge());
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(new NewContactsAdapterBridge());
  }
}
```

The Android 2.x edition uses `ContactsContract` to find just the mobile numbers:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.database.Cursor;
import android.provider.ContactsContract.Contacts;
import android.provider.ContactsContract.CommonDataKinds.Phone;
import android.widget.SpinnerAdapter;
import android.widget.SimpleCursorAdapter;

class NewContactsAdapterBridge extends ContactsAdapterBridge {
  SpinnerAdapter buildPhonesAdapter(Activity a) {
    String[] PROJECTION=new String[] { Contacts._ID,
                                       Contacts.DISPLAY_NAME,
                                       Phone.NUMBER
                                     };
    String[] ARGS={String.valueOf(Phone.TYPE_MOBILE)};
    Cursor c=a.managedQuery(Phone.CONTENT_URI,
                            PROJECTION, Phone.TYPE+"=?",
                            ARGS, Contacts.DISPLAY_NAME);

    SimpleCursorAdapter adapter=new SimpleCursorAdapter(a,
                                    android.R.layout.simple_spinner_item,
                                    c,
                                    new String[] {
                                      Contacts.DISPLAY_NAME
                                    },
                                    new int[] {
                                      android.R.id.text1
                                    });

    adapter.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);

    return(adapter);
  }
}
```

... while the Android 1.x edition uses the older `Contacts` provider to find the mobile numbers:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.database.Cursor;
import android.provider.Contacts;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.widget.SimpleCursorAdapter;
import android.widget.SpinnerAdapter;

@SuppressWarnings("deprecation")
class OldContactsAdapterBridge extends ContactsAdapterBridge {
  SpinnerAdapter buildPhonesAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts.Phones._ID,
                                        Contacts.Phones.NAME,
                                        Contacts.Phones.NUMBER
                                      };
    String[] ARGS={String.valueOf(Contacts.Phones.TYPE_MOBILE)};
    Cursor c=a.managedQuery(Contacts.Phones.CONTENT_URI,
                            PROJECTION,
                            Contacts.Phones.TYPE+"=?", ARGS,
                            Contacts.Phones.NAME);

    SimpleCursorAdapter adapter=new SimpleCursorAdapter(a,
                                    android.R.layout.simple_spinner_item,
                                    c,
                                    new String[] {
                                      Contacts.Phones.NAME
                                    },
                                    new int[] {
                                      android.R.id.text1
                                    });

    adapter.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);

    return(adapter);
  }
}
```

For more details on how those providers work, please see the chapter on contacts.

The activity then loads up the `Spinner` with the appropriate list of contacts. When the user taps the Send button, the `sendTheMessage()` method is invoked (courtesy of the `android:onClick` attribute in the layout). That method looks at the radio buttons, sees which one is selected, and routes the text message accordingly:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.telephony.SmsManager;
import android.view.View;
import android.widget.EditText;
import android.widget.RadioGroup;
import android.widget.Spinner;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public class Sender extends Activity {
  Spinner contacts=null;
  RadioGroup means=null;
  EditText msg=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    contacts=(Spinner)findViewById(R.id.spinner);

    contacts.setAdapter(ContactsAdapterBridge
                        .INSTANCE
                        .buildPhonesAdapter(this));

    means=(RadioGroup)findViewById(R.id.means);
    msg=(EditText)findViewById(R.id.msg);
  }

  public void sendTheMessage(View v) {
    Cursor c=(Cursor)contacts.getSelectedItem();

    if (means.getCheckedRadioButtonId()==R.id.client) {
      Intent sms=new Intent(Intent.ACTION_SENDTO,
                            Uri.parse("smsto:"+c.getString(2)));

      sms.putExtra("sms_body", msg.getText().toString());

      startActivity(sms);
    }
    else {
      SmsManager
        .getDefault()
        .sendTextMessage(c.getString(2), null,
                         msg.getText().toString(),
                         null, null);
    }
  }
}
```

SMS Sending Limitations

Apps running on Android 1.x and 2.x devices are limited to sending 100 SMS messages an hour, before the user starts getting prompted with each SMS message request to confirm that they do indeed wish to send it.

Apps running on Android 4.x devices, the limits are now 30 SMS messages in 30 minutes, according to some source code analysis by Al Sutton.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Monitoring and Receiving SMS

For the purposes of this section, "monitoring" refers to the ability to inspect incoming SMS messages, including reading their contents. In contrast, "receiving" SMS messages is actually consuming the message and storing it somewhere for the user to use.

As it turns out, "monitoring" and "receiving" are much the same thing prior to Android 4.4, but are significantly different in the new API made available in Android 4.4

The Undocumented, Unsupported, Pre-Android 4.4 Way

It is possible for an application to monitor or receive an incoming SMS message... if you are willing to listen on the undocumented `android.provider.Telephony.SMS_RECEIVED` broadcast `Intent`. That is sent by Android whenever an SMS arrives, and it is up to an application to implement a `BroadcastReceiver` to respond to that `Intent` and do something with the message. The Android open source project has such an application — Messaging — and device manufacturers can replace it with something else.

Note that to listen for this broadcast, your app must hold the `RECEIVE_SMS` permission.

The `BroadcastReceiver` can then turn around and use the `SmsMessage` class, in the `android.telephony` package, to get at the message itself, through the following undocumented recipe:

1. Given the received `Intent` (`intent`), call `intent.getExtras().get("pdus")` to get an `Object` array representing the raw portions of the message
2. For each of those "pdus" objects, call `SmsMessage.createFromPdu()` to convert the `Object` into an `SmsMessage` — though to make this work, you need to cast the `Object` to a `byte` array as part of passing it to the `createFromPdu()` static method

The resulting `SmsMessage` object gets you access to the text of the message, the sending phone number, etc.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The SMS_RECEIVED broadcast Intent is broadcast a bit differently than most others in Android. It is an "ordered broadcast", meaning the Intent will be delivered to one BroadcastReceiver at a time. This has two impacts of note:

- In your receiver's <intent-filter> element, you can have an android:priority attribute. Higher priority values get access to the broadcast Intent earlier than will lower priority values. The standard Messaging application has the default priority (undocumented, appears to be 0 or 1), so you can arrange to get access to the SMS before the application does.
- Your BroadcastReceiver can call abortBroadcast() on itself to prevent the Intent from being broadcast to other receivers of lower priority. In effect, this causes your receiver to consume the SMS — the Messaging application will not receive it. So, aborting the broadcast means that your app chose to "receive" the SMS; not aborting the broadcast means that your app is merely "monitoring" the SMS messages that come in.

However, just because the Messaging application has the default priority does not mean all SMS clients will, and so you cannot reliably intercept SMS messages this way. That, plus the undocumented nature of all of this, means that applications you write to receive SMS messages are likely to be fragile in production, breaking on various devices due to device manufacturer-installed apps, third-party apps, or changes to Android itself... such as the changes that came about in Android 4.4.

The Android 4.4+ Way: Monitoring SMS

The code described above still works on Android 4.4, though the formerly-hidden android.provider.Telephony class is now part of the SDK.

The biggest difference, though, is that even if you call abortBroadcast(), the user's chosen SMS messaging client will still receive the message. It is not possible for an app listening for SMS_RECEIVED broadcasts to prevent the user's chosen SMS messaging client from receiving those same messages. This is a substantial change, one that will break or make obsolete many Android applications.

Regardless, if monitoring SMS fits your needs, SMS_RECEIVED can do it.

So, for example, the SMS/Monitor sample project implements a BroadcastReceiver for SMS_RECEIVED, one with slightly elevated priority:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.sms.monitor"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"/>

  <uses-permission android:name="android.permission.RECEIVE_SMS"/>

  <uses-sdk
    android:minSdkVersion="3"
    android:targetSdkVersion="19"/>

  <application
    android:icon="@drawable/cw"
    android:label="@string/app_name">
    <receiver
      android:name="Monitor"
      android:permission="android.permission.BROADCAST_SMS">
      <intent-filter android:priority="2">
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
      </intent-filter>
    </receiver>

    <activity
      android:name="BootstrapActivity"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

You will notice that the `BroadcastReceiver` not only has the slightly-elevated priority (`android:priority="2"`), but also a required permission (`android:permission="android.permission.BROADCAST_SMS"`). Only apps that hold this permission can send this broadcast in a way that will be picked up by the receiver. Since this permission can only be held by the device firmware, you are protected from "spoof" SMS messages from rogue apps on the device, sending the `SMS_RECEIVED` themselves.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The app also has a do-nothing activity, solely there to activate the manifest-registered `BroadcastReceiver`, which will not work until some component of the app is manually started.

The bulk of the business logic — what little there is of it — lies in the `Monitor` class that is the `BroadcastReceiver`:

```
package com.commonsware.android.sms.monitor;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.telephony.SmsMessage;
import android.util.Log;

public class Monitor extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    Object[] rawMsgs=(Object[])intent.getExtras().get("pdus");

    for (Object raw : rawMsgs) {
      SmsMessage msg=SmsMessage.createFromPdu((byte[])raw);

      if (msg.getMessageBody().toUpperCase().contains("SEKRIT")) {
        Log.w("SMS:"+msg.getOriginatingAddress(),
              msg.getMessageBody());

        abortBroadcast();
      }
    }
  }
}
```

Here, we retrieve the raw messages from the `Intent` extra, iterate over them, and convert each to an `SmsMessage`. Those that have the magic word in their message body will result in the message being dumped to LogCat, plus the broadcast is aborted. On Android 4.3 and below, this will prevent lower-priority receivers from receiving the SMS. On Android 4.4, the abort request is ignored.

The Android 4.4+ Way: Receiving SMS

Receiving SMS messages, on Android 4.4+, means that you are implementing an SMS client application, one the user might be willing to set as their default SMS client application in Settings. There are other sorts of apps that may temporarily want to be the default SMS client, such as a backup/restore utility, as only the default SMS client will be able to work with the SMS `ContentProvider` suite, such as the inbox.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Receiving the Broadcasts

The default SMS client should be able to handle both SMS and MMS. This is a problem, as while supporting SMS is poorly documented, supporting MMS has almost no documentation whatsoever. However, unless the default SMS client handles MMS, nobody else can (at least, while saving MMS details to the `ContentProvider` suite.

Hence, Google is expecting you to have two `BroadcastReceivers` registered in the manifest: one for SMS and one for MMS. Unfortunately, these cannot readily be combined into a single receiver, because each has its own permission requirement:

- the SMS receiver should require senders to hold `BROADCAST_SMS`
- the MMS receiver should require senders to hold `BROADCAST_WAP_PUSH`

In practice, probably both are held by the OS component that is sending these broadcasts in response to incoming messages of either type. In principle, though, they could be separate, and an individual `<receiver>` can only specify one such permission.

The Android documentation illustrates the `<receiver>` elements that Google expects your SMS client application to have:

```
<!-- BroadcastReceiver that listens for incoming SMS messages -->
<receiver android:name=".SmsReceiver"
        android:permission="android.permission.BROADCAST_SMS">
    <intent-filter>
        <action android:name="android.provider.Telephony.SMS_DELIVER" />
    </intent-filter>
</receiver>

<!-- BroadcastReceiver that listens for incoming MMS messages -->
<receiver android:name=".MmsReceiver"
    android:permission="android.permission.BROADCAST_WAP_PUSH">
    <intent-filter>
        <action android:name="android.provider.Telephony.WAP_PUSH_DELIVER" />
        <data android:mimeType="application/vnd.wap.mms-message" />
    </intent-filter>
</receiver>
```

Notice that the MMS receiver has both an `<action>` and a `<data>` element in its `<intent-filter>`, which is rather unusual.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

On the SMS side, the `Intent` you receive should be the same as the `Intent` you would receive for the `SMS_RECEIVED` broadcast, where you can decode the message(s) and deal with them as you see fit. On the MMS side... there is little documentation.

Other Expectations

Google expects the default SMS client to be able to handle `ACTION_SEND` and `ACTION_SENDTO` for relevant schemes:

```
<!-- Activity that allows the user to send new SMS/MMS messages -->
<activity android:name=".ComposeSmsActivity" >
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <action android:name="android.intent.action.SENDTO" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="sms" />
        <data android:scheme="smsto" />
        <data android:scheme="mms" />
        <data android:scheme="mmsto" />
    </intent-filter>
</activity>
```

That may not be terribly surprising. What *is* surprising is that Google also expects you to have an exported *service* for handling "quick response" requests. These requests come when the user receives a phone call and taps on an icon to reply with a text message, rather than accept the call. In those cases, Android will invoke a service in the default SMS client, with an action of `android.intent.action.RESPOND_VIA_MESSAGE`. The `Intent` that you receive in `onStartCommand()` (or `onHandleIntent()`, if you elect to use an `IntentService`) will have an `EXTRA_TEXT` and optionally an `EXTRA_SUBJECT` as extras, representing the message to be sent. The `Uri` in the `Intent` will indicate the intended recipient of the message. Your job is to use `SmsManager` to actually send the message.

The Android documentation cites this as the relevant `<service>` element:

```
<!-- Service that delivers messages from the phone "quick response" -->
<service android:name=".HeadlessSmsSendService"
         android:permission="android.permission.SEND_RESPOND_VIA_MESSAGE"
         android:exported="true" >
    <intent-filter>
        <action android:name="android.intent.action.RESPOND_VIA_MESSAGE" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="sms" />
        <data android:scheme="smsto" />
        <data android:scheme="mms" />
        <data android:scheme="mmsto" />
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    </intent-filter>
</service>
```

Note:

- The `<service>` requires that the sender have the `SEND_RESPOND_VIA_MESSAGE` permission, to reduce spoofing
- The `android:exported="true"` shown in the sample should be superfluous, as since the `<service>` has an `<intent-filter>`, it should be exported by default
- The `<category>`, and possibly the `<data>`, elements may be erroneous... and since the author cannot find anything in the OS that uses `RESPOND_VIA_MESSAGE`, the author cannot validate that these elements should be here or represent copy-and-paste errors in the documentation

Handling Both Receive Options

If you want to support receiving SMS using both the legacy approach and the Android 4.4+ approach, you can have two `BroadcastReceiver` implementations, one for `android.provider.Telephony.SMS_RECEIVED` and one for `android.provider.Telephony.SMS_DELIVER`. However, you will only need the latter one on Android 4.4, and by default you would receive both broadcasts.

To handle that, you can define a boolean resource in the `res/values-v19/` directory (e.g., `isPreKitKat`) to be `false`, with a default definition in `res/values/` of `true` for the same resource. Then, in your manifest, you can have `android:enabled="@bool/isPreKitKat"` on your `SMS_RECEIVED` `<receiver>` element. This will only enable this component on API Level 18 and below, disabling it on API Level 19+.

You can also define a counterpart resource for the positive case (e.g., `@bool/isKitKat`), and use that to selectively enable the SMS and MMS receivers, if desired.

The SMS Inbox

Many users keep their text messages around, at least for a while. These are stored in an "inbox", represented by a `ContentProvider`. How you work with this `ContentProvider` — or if you can work with it at all, varies upon whether you are running on Android 4.4+ or not.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Undocumented, Unsupported, Pre-Android 4.4 Way

When perusing the Internet, you will find various blog posts and such referring to the SMS inbox `ContentProvider`, represented by the `content://sms/inbox` `Uri`.

This `ContentProvider` is undocumented and is not part of the Android SDK, because it is not part of the Android OS.

Rather, this `ContentProvider` is used by the aforementioned Messaging application, for storing saved SMS messages. And, as noted, this application may or may not exist on any given Android device. If a device manufacturer replaces Messaging with their own application, there may be nothing on that device that responds to that `Uri`, or the schemas may be totally different. Plus, Android may well change or even remove this `ContentProvider` in future editions of Android.

For all those reasons, developers should not be relying upon this `ContentProvider`.

The Android 4.4+ Way

Android 4.4 has exposed a series of `ContentProviders`, in the `android.provider.Telephony` namespace, for storing SMS and MMS messages. These include:

- the `Inbox` for received messages
- the `Outbox` for a log of sent messages
- the `Draft` for messages that were written but have not yet been sent
- etc.

Some are duplicated, such as separate providers for the SMS inbox versus the MMS inbox. Some are distinct, such as `Sms.Conversations` and `Mms.Rate`.

All are largely undocumented.

The user's chosen default SMS client can write to these providers. Apps with `READ_SMS` permission should be able to read from them.

Asking to Change the Default

There are many areas in Android where the user must do two things to use an app:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. Install the app (from the Play Store or elsewhere)
2. Go into Settings (or sometimes elsewhere) and indicate that a certain capability of the newly-installed app should become active

You see this with app widgets, input method editors, device administrators, and many others.

On Android 4.4+, you also see this with SMS/MMS clients. Devices usually ship with one. If the user wants a replacement, the user must indicate in Settings that this new SMS/MMS client should be the default, so it can write to the SMS/MMS `ContentProvider` suite.

Your app can determine what the default client is by calling `getDefaultSmsPackage()` on the `Telephony.Sms` class. This will return the package name of the current default client.

If this is not your package, and you would like the user to make you the default, you can start an activity to request this change:

```
Intent i = new Intent(Sms.Intents.ACTION_CHANGE_DEFAULT);
i.putExtra(Sms.Intents.EXTRA_PACKAGE_NAME, getPackageName());
startActivity(i);
```

The `EXTRA_PACKAGE_NAME` will trigger the UI to ask the user if the user wishes to change the current default to your package (versus anything else on the device that might also be a possible SMS/MMS client).

Hence, the recommended flow for a backup/restore app is to:

- Make note of the current default, via `getDefaultSmsPackage()`
- Request to the user to make you the default, via `ACTION_CHANGE_DEFAULT`
- Confirm that they did this, via `getDefaultSmsPackage()`
- If they did, do your backup or restore work
- Request to the user to restore the original default, via `ACTION_CHANGE_DEFAULT`

SMS and the Emulator

The “Emulator Control” view in DDMS allows you to send fake SMS messages to a running emulator. This is very useful for light testing.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You can also send fake SMS messages to an emulator via the emulator console. This can be accessed via `telnet`, where the console is available on `localhost` on your development machine, via the port number that appears in the title bar of your emulator window (e.g., `5554`). In the `telnet` session, you can enter `sms send [sendingNumber> <txt>`, replacing `<sendingNumber>` with the phone number of the pretend sender of the SMS, and replacing `<txt>` with the text message itself.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

NFC

NFC, courtesy of high-profile boosters like Google Wallet, is poised to be a significant new capability in Android devices. While at the time of this writing, only a handful of Android devices have NFC built in, other handsets are slated to be NFC-capable in the coming months. Google is hoping that developers will write NFC-aware applications to help further drive adoption of this technology by device manufacturers.

This, of course, raises the question: what is NFC? Besides being where the Green Bay Packers play, that is?

(For those of you from outside of the United States, that was an American football joke. We now return you to your regularly-scheduled chapter.)

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapters on broadcast `Intents` and services.

What Is NFC?

NFC stands for Near-Field Communications. It is a wireless standard for data exchange, aimed at very short range transmissions — on the order of a couple of centimeters. NFC is in wide use today, for everything from credit cards to passports. Typically, the NFC data exchange is for simple data — contact information, URLs, and the like.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In particular, NFC tends to be widely used where one side of the communications channel is "passive", or unpowered. The other side (the "initiator") broadcasts a signal, which the passive side converts into power enough to send back its response. As such, NFC "tags" containing such passive targets can be made fairly small and can be embedded in a wide range of containers, from stickers to cards to hats.

The objective is "low friction" interaction — no pairing like with Bluetooth, no IP address shenanigans as with WiFi. The user just taps and goes.

… Compared to RFID?

NFC is often confused with or compared to RFID. It is simplest to think of RFID as being an umbrella term, under which NFC falls. Not every RFID technology is NFC, but many things that you hear of being "RFID" may actually be NFC-compliant devices or tags.

… Compared to QR Codes?

In many places, NFC will be used in ways you might consider using QR codes. For example, a restaurant could use either technology, or both, on a sign to lead patrons to the restaurant's Yelp page, as a way of soliciting reviews. Somebody with a capable device could either tap the NFC tag on the sign to bring up Yelp or take a picture of the QR code and use that to bring up Yelp.

NFC's primary advantage over QR codes is that it requires no user intervention beyond physically moving their device in close proximity to the tag. QR codes, on the other hand, require the user to launch a barcode scanning application, center the barcode in the viewfinder, and then get the results. The net effect is that NFC will be faster.

QR's advantages include:

1. No need for any special hardware to generate the code, as opposed to needing a tag and something to write information into the tag for NFC
2. The ability to display QR codes in distant locations (e.g., via Web sites), whereas NFC requires physical proximity

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To NDEF, Or Not to NDEF

RFID is a concept, not a standard. As such, different vendors created their own ways of structuring data on these tags or chips, making one vendor's tags incompatible with another vendor's readers or writers. While various standards bodies, like ISO, have gotten involved, it's still a bit of a rat's nest of conflicting formats and approaches.

The NFC offshoot of RFID has had somewhat greater success in establishing standards. NFC itself is an ISO and ECMA standard, covering things like transport protocols and transfer speeds. And a consortium called the NFC Forum created NDEF — the NFC Data Exchange Format — for specifying the content of tags.

However, not all NFC tags necessarily support NDEF. NDEF is much newer than NFC, and so lots of NFC tags are out in the wild that were distributed before NDEF even existed.

You can roughly divide NFC tags into three buckets:

- Those that support NDEF "out of the box"
- Those that can be "formatted" as NDEF
- Those that use other content schemes

Android has some support for non-NDEF tags, such as the MIFARE Classic. However, the hope and expectation going forward is that NFC tags will coalesce around NDEF.

NDEF, as it turns out, maps neatly to Android's `Intent` system, as you will see as we proceed through this chapter.

NDEF Modalities

Most developers interested in NFC will be interested in reading NFC tags and retrieving the NDEF data off of them. In Android, tapping an NDEF tag with an NFC-capable device will trigger an activity to be started, based on a certain `IntentFilter`.

Some developers will be interested in writing to NFC tags, putting URLs, vCards, or other information on them. This may or may not be possible for any given tag.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

And while the "traditional" thinking around NFC has been that one side of the communication is a passive tag, Android will help promote the "peer-to-peer" approach — having two Android devices exchange data via NFC and NDEF. Basically, putting the two devices back-to-back will cause each to detect the other device's "tag", and each can read and write to the other via this means. This is referred to as "Android Beam" and will be discussed later in this chapter.

Of course, all of these are only available on hardware. At the present time, there is no emulator for NFC, nor any means of accessing a USB NFC reader or writer from the emulator.

NDEF Structure and Android's Translation

NDEF is made up of messages, themselves made up of a series of records. From Android's standpoint, each tag consists of one such message.

Each record consists of a binary (`byte` array) payload plus metadata to describe the nature of the payload. The metadata primarily consists of a type and a subtype. There are quite a few combinations of these, but the big three for new Android NFC uses are:

- A type of `TNF_WELL_KNOWN` and a subtype of `RTD_TEXT`, indicating that the payload is simply plain text
- A type of `TNF_WELL_KNOWN` and a subtype of `RTD_URI`, indicating that the payload is a URI, such as a URL to a Web page
- A type of `TNF_MIME_MEDIA`, where the subtype is a standard MIME type, indicating that the payload is of that MIME type

When Android scans an NDEF tag, it will use this information to construct a suitable `Intent` to use with `startActivity()`. The action will be `android.nfc.action.NDEF_DISCOVERED`, to distinguish the scanned-tag case from, say, something simply asking to view some content. The MIME type in the Intent will be `text/plain` for the first scenario above or the supplied MIME type for the third scenario above. The data (`Uri`) in the Intent will be the supplied URI for the second scenario above. Once constructed, Android will invoke `startActivity()` on that `Intent`, bringing up an activity or an activity chooser, as appropriate.

NFC-capable Android devices have a Tags application pre-installed that will handle any NFC tag not handled by some other app. So, for example, an NDEF tag with an

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

HTTP URL will fire up the Tags application, which in turn will allow the user to open up a Web browser on that URL.

The Reality of NDEF

The enthusiasm that some have with regards to Android and NFC technology needs to be tempered by the reality of NDEF, NFC tags in general, and Android's support for NFC. It is easy to imagine all sorts of possibilities that may or may not be practical when current limitations are reached.

Some Tags are Read-Only

Some tags come "from the factory" read-only. Either you arrange for the distributor to write data onto them (e.g., blast a certain URL onto a bunch of NFC stickers to paste onto signs), or they come with some other pre-established data. Touchatag, for example, distributes NFC tags that have Touchatag URLs on them — they then help you set up redirects from their supplied URL to ones you supply.

While these tags will be of interest to consumers and businesses, they are unlikely to be of interest to Android developers, since their use cases are already established and typically do not need custom Android application support. Android developers seeking customizable tags will want ones that are read-write, or at least write-once.

Some Tags Can't Be Read-Only

Conversely, some tags lack any sort of read-only flag. An ideal tag for developers is one that is write-once: putting an NDEF message on the tag and flagging it read-only in one operation. Some tags do not support this, or making the tag read-only at any later point. The MIFARE Classic 1K tag is an example — while technically it can be made read-only, it requires a key known only to the tag manufacturer.

Some Tags Need to be Formatted

The MIFARE Classic 1K NFC tag is NDEF-capable, but must be "formatted" first, supplying the initial NDEF message contents. You have the option of formatting it read-write or read-only (turning the Classic 1K a write-once tag).

This is not a problem — in fact, the write-once option may be compelling. However, it is something to keep in mind.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Also, note that the MIFARE Classic 1K, while it can be formatted as NDEF, uses a proprietary protocol "under the covers". Not all Android devices will support the Classic 1K, as the device manufacturers elect not to pay the licensing fee. Where possible, try to stick to tags that are natively NDEF-compliant (so-called "NFC Forum Tag Types 1–4").

Tags Have Limited Storage

The "1K" in the name "MIFARE Classic 1K" refers to the amount of storage on the tag: 1 kilobyte of information.

And that's far larger than other tags, such as the MIFARE Ultralight C, some of which have ~64 bytes of storage.

Clearly, you will not be writing an MP3 file or JPEG photo to these tags. Rather, the tags will tend to either be a "launcher" into something with richer communications (e.g., URL to a Web site) or will use the sorts of data you may be used to from QR codes, such as a vCard or iCalendar for contact and event data, respectively.

NDEF Data Structures Are Documented Elsewhere

The Android developer documentation is focused on the Android classes related to NFC and on the `Intent` mechanism used for scanned tags. It does not focus on the actual structure of the payloads.

For `TNF_MIME_MEDIA` and `RTD_TEXT`, the payload is whatever you want. For `RTD_URI`, however, the byte array has a bit more structure to it, as the NDEF specification calls for a single byte to represent the URI prefix (e.g., `http://www.` versus `http://` versus `https://www.`). The objective, presumably, is to support incrementally longer URLs on tags with minuscule storage. Hence, you will need to convert your URLs into this sort of byte array if you are writing them out to a tag.

Generally speaking, the rules surrounding the structure of NDEF messages and records is found at the NFC Forum site.

Tag and Device Compatibility

Different devices will have different NFC chipsets. Not all NFC chipsets can read and write all tags. The expectation is that NDEF-formatted tags will work on all devices,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

but if you wander away from that, things get dicier. For example, NXP's Mifare Classic tag can only be read and written by NXP's NFC chip.

This is increasingly a challenge for Android developers, as a Broadcom NFC chip is becoming significantly more popular. Many new major Android devices, such as the Samsung Galaxy S4, the Nexus 4, the Nexus 10, and the 2013/2nd generation version of the Nexus 7, all use the Broadcom chip. Those devices are incompatible with the Mifare tags, such as the popular Mifare Classic 1K.

That is because NXP is the maker of the Mifare Classic series, and those tags broke the NFC Forum's standards to create a tag that was NXP-specific.

Right now, NTAG203 and Topaz tags (like the Topaz 512), are likely candidate tags that will work across all NFC-capable Android devices, due to their adherence to NFC standard protocols.

Sources of Tags

NFC tags are not the sort of thing you will find on your grocer's shelves. In fact, few, if any, mainstream firms sell them today.

Here are some online sites from which you can order rewritable NFC tags, listed here in alphabetical order:

1. Andytags
2. Buy NFC Stickers
3. Buy NFC Tags
4. Smartcard Focus
5. tagstand

Note that not all may ship to your locale.

Writing to a Tag

So, let's see what it takes to write an NDEF message to a tag, formatting it if needed. The code samples shown in this chapter are from the `NFC/URLTagger` sample application. This application will set up an activity to respond to `ACTION_SEND` activity `Intents`, with an eye towards receiving a URL from a browser, then waiting for a tag and writing the URL to that tag. The idea is that this sort of application

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

could be used by non-technical people to populate tags containing URLs to their company's Web site, etc.

Getting a URL

First, we need to get a URL from the browser. As we saw in the chapter on integration, the standard Android browser uses `ACTION_SEND` of `text/plain` contents when the user chooses the "Share Page" menu. So, we have one activity, `URLTagger`, that will respond to such an `Intent`:

```
    <activity
      android:name="URLTagger"
      android:label="@string/app_name">
      <intent-filter android:label="@string/app_name">
        <action android:name="android.intent.action.SEND"/>

        <data android:mimeType="text/plain"/>

        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
    </activity>
```

Of course, lots of other applications support `ACTION_SEND` of `text/plain` contents that are not URLs. A production-grade version of this application would want to validate the `EXTRA_TEXT` `Intent` extra to confirm that, indeed, this is a URL, before putting in an NDEF message claiming that it is a URL.

Detecting a Tag

When the user shares a URL with our application, our activity is launched. At that point, we need to go into "detect a tag" mode – the user should then tap their device to a tag, so we can write out the URL.

First, in `onCreate()`, we get access to the `NfcAdapter`, which is our gateway to much of the NFC functionality in Android:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    nfc=NfcAdapter.getDefaultAdapter(this);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We use a boolean data member — `inWriteMode` — to keep track of whether or not we are set up to write to a tag. Initially, of course, that is set to be `false`. Hence, when we are first launched, by the time we get to `onResume()`, we can go ahead and register our interest in future tags:

```
@Override
public void onResume() {
  super.onResume();

  if (!inWriteMode) {
    IntentFilter discovery=new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
    IntentFilter[] tagFilters=new IntentFilter[] { discovery };
    Intent i=new Intent(this, getClass())
              .addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP|
                        Intent.FLAG_ACTIVITY_CLEAR_TOP);
    PendingIntent pi=PendingIntent.getActivity(this, 0, i, 0);

    inWriteMode=true;
    nfc.enableForegroundDispatch(this, pi, tagFilters, null);
  }
}
```

When an NDEF-capable tag is within signal range of the device, Android will invoke `startActivity()` for the `NfcAdapter.ACTION_TAG_DISCOVERED` `Intent` action. However, it can do this in one of two ways:

- Normally, it will use a chooser (via `Intent.createChooser()`) to allow the user to pick from any activities that claim to support this action.
- The foreground application can request via `enableForegroundDispatch()` for it to handle all tag events while it is in the foreground, superseding the normal `startActivity()` flow. In this case, while Android still will invoke an activity, it will be our activity, not any other one.

We want the second approach right now, so the next tag brought in range is the one we will try writing to.

To do that, we need to create an array of `IntentFilter` objects, identifying the NFC-related actions that we want to capture in the foreground. In this case, we only care about `ACTION_TAG_DISCOVERED` – if we were supporting non-NDEF NFC tags, we might also need to watch for `ACTION_TECH_DISCOVERED`.

We also need a `PendingIntent` identifying the activity that should be invoked when such a tag is encountered while we are in the foreground. Typically, this will be the current activity. By adding `FLAG_ACTIVITY_SINGLE_TOP` and

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

FLAG_ACTIVITY_CLEAR_TOP to the Intent as flags, we ensure that our current specific instance of the activity will be given control again via onNewIntent().

Armed with those two values, we can call enableForegroundDispatch() on the NfcAdapter to register our request to process tags via the current activity instance.

In onPause(), if the activity is finishing, we call disableForegroundDispatch() to undo the work done in onResume():

```
  @Override
  public void onPause() {
    if (isFinishing()) {
      nfc.disableForegroundDispatch(this);
      inWriteMode=false;
    }

    super.onPause();
  }
```

We have to see if we are finishing, because even though our activity never leaves the screen, Android still calls onPause() and onResume() as part of delivering the Intent to onNewIntent(). Our approach, though, has flaws — if the user presses HOME, for example, we never disable the NFC dispatch logic. A production-grade application would need to handle this better.

For any of this code to work, we need to hold the NFC permission via an appropriate line in the manifest:

```
<uses-permission android:name="android.permission.NFC"/>
```

Also note that if you have several activities that the user can reach while you are trying to also capture NFC tag events, you will need to call enableForegroundDispatch() in each activity — it's a per-activity request, not a per-application request.

Reacting to a Tag

Once the user brings a tag in range, onNewIntent() will be invoked with the ACTION_TAG_DISCOVERED Intent action:

```
  @Override
  protected void onNewIntent(Intent intent) {
    if (inWriteMode &&
        NfcAdapter.ACTION_TAG_DISCOVERED.equals(intent.getAction())) {
      Tag tag=intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      byte[] url=buildUrlBytes(getIntent().getStringExtra(Intent.EXTRA_TEXT));
      NdefRecord record=new NdefRecord(NdefRecord.TNF_WELL_KNOWN,
                                       NdefRecord.RTD_URI,
                                       new byte[] {}, url);
      NdefMessage msg=new NdefMessage(new NdefRecord[] {record});

      new WriteTask(this, msg, tag).execute();
    }
  }
```

If we are in write mode and the delivered `Intent` is indeed an `ACTION_TAG_DISCOVERED` one, we can get at the `Tag` object associated with the user's NFC tag via the `NfcAdapter.EXTRA_TAG` `Parcelable` extra on the `Intent`.

Writing an NDEF message to the tag, therefore, is a matter of crafting the message and actually writing it. An NDEF message consists of one or more records (though, typically, only one record is used), with each record wrapping around a byte array of payload data.

Getting the Shared URL

We did not do anything to get the URL out of the `Intent` back in `onCreate()`, when our activity was first started up. Now, of course, we need that URL. You might think it is too late to get it, since our activity was effectively started again due to the tag and `onNewIntent()`.

However, `getIntent()` on an `Activity` always returns the `Intent` used to create the activity in the first place. The `getIntent()` value is not replaced when `onNewIntent()` is called.

Hence, as part of the `buildUrlBytes()` method to create the binary payload, we can go and call `getIntent().getStringExtra(Intent.EXTRA_TEXT)` to retrieve the URL.

Creating the Byte Array

Given the URL, we need to convert it into a byte array suitable for use in a `TNF_WELL_KNOWN`, `RTD_URI` NDEF record. Ordinarily, you would just call `toByteArray()` on the `String` and be done with it. However, the byte array we need uses a single byte to indicate the URL prefix, with the rest of the byte array for the characters after this prefix.

This is efficient. This is understandable. This is annoying.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

First, we need the roster of prefixes, defined in `URLTagger` as a static data member cunningly named `PREFIXES`:

```
  static private final String[] PREFIXES={"http://www.", "https://www.",
                                          "http://", "https://",
                                          "tel:", "mailto:",
                                          "ftp://anonymous:anonymous@",
                                          "ftp://ftp.", "ftps://",
                                          "sftp://", "smb://",
                                          "nfs://", "ftp://",
                                          "dav://", "news:",
                                          "telnet://", "imap:",
                                          "rtsp://", "urn:",
                                          "pop:", "sip:", "sips:",
                                          "tftp:", "btspp://",
                                          "btl2cap://", "btgoep://",
                                          "tcpobex://",
                                          "irdaobex://",
                                          "file://", "urn:epc:id:",
                                          "urn:epc:tag:",
                                          "urn:epc:pat:",
                                          "urn:epc:raw:",
                                          "urn:epc:", "urn:nfc:"};
```

Then, in `buildUrlBytes()`, we need to find the prefix (if any) and use it:

```
  private byte[] buildUrlBytes(String url) {
    byte prefixByte=0;
    String subset=url;
    int bestPrefixLength=0;

    for (int i=0;i<PREFIXES.length;i++) {
      String prefix = PREFIXES[i];

      if (url.startsWith(prefix) && prefix.length() > bestPrefixLength) {
        prefixByte=(byte)(i+1);
        bestPrefixLength=prefix.length();
        subset=url.substring(bestPrefixLength);
      }
    }

    final byte[] subsetBytes = subset.getBytes();
    final byte[] result = new byte[subsetBytes.length+1];

    result[0]=prefixByte;
    System.arraycopy(subsetBytes, 0, result, 1, subsetBytes.length);

    return(result);
  }
```

We iterate over the `PREFIXES` array and find a match, if any, and the best possible match if there is more than one. If there is a match, we record the NDEF value for

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the first byte (our `PREFIXES` index plus one) and create a subset string containing the characters after the prefix. If there is no matching prefix, the prefix byte is `0` and we will include the full URL.

Given that, we construct a byte array containing our prefix byte in the first slot, and the rest taken up by the byte array of the subset of our URL.

Creating the NDEF Record and Message

Given the result of `buildUrlBytes()`, our `onNewIntent()` implementation creates a `TNF_WELL_KNOWN`, `RTD_URI` `NdefRecord` object, and pours that into an `NdefMessage` object.

The third parameter to the `NdefRecord` constructor is a byte array representing the optional "ID" of this record, which is not necessary here.

Finally, we delegate the actual writing to a `WriteTask` subclass of `AsyncTask`, as writing the `NdefMessage` to the `Tag` is... interesting.

Writing to a Tag

Here is the aforementioned `WriteTask` static inner class:

```
  static class WriteTask extends AsyncTask<Void, Void, Void> {
    Activity host=null;
    NdefMessage msg=null;
    Tag tag=null;
    String text=null;

    WriteTask(Activity host, NdefMessage msg, Tag tag) {
      this.host=host;
      this.msg=msg;
      this.tag=tag;
    }

    @Override
    protected Void doInBackground(Void... arg0) {
      int size=msg.toByteArray().length;

      try {
        Ndef ndef=Ndef.get(tag);

        if (ndef==null) {
          NdefFormatable formatable=NdefFormatable.get(tag);

          if (formatable!=null) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
          try {
            formatable.connect();

            try {
              formatable.format(msg);
            }
            catch (Exception e) {
              text="Tag refused to format";
            }
          }
          catch (Exception e) {
            text="Tag refused to connect";
          }
          finally {
            formatable.close();
          }
        }
        else {
          text="Tag does not support NDEF";
        }
      }
      else {
        ndef.connect();

        try {
          if (!ndef.isWritable()) {
            text="Tag is read-only";
          }
          else if (ndef.getMaxSize()<size) {
            text="Message is too big for tag";
          }
          else {
            ndef.writeNdefMessage(msg);
          }
        }
        catch (Exception e) {
          text="Tag refused to connect";
        }
        finally {
          ndef.close();
        }
      }
    }
    catch (Exception e) {
      Log.e("URLTagger", "Exception when writing tag", e);
      text="General exception: "+e.getMessage();
    }

    return(null);
  }

  @Override
  protected void onPostExecute(Void unused) {
    if (text!=null) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        Toast.makeText(host, text, Toast.LENGTH_SHORT).show();
      }

      host.finish();
    }
  }
```

In `doInBackground()`, after making note of how big the message is in bytes, we first try to get the `Ndef` aspect of the `Tag` object, by calling the static `get()` method on the `Ndef` class. If the tag is an NDEF tag, this should return an `Ndef` instance. If it does not, we try to get an `NdefFormatable` aspect by calling `get()` on the `NdefFormatable` class. If the tag is not NDEF now but can be formatted as NDEF, this should give us an `NdefFormatable` object. If both aspect attempts fail, we bail out, displaying a `Toast` to let the user know that while the tag they used is NFC, it is not NDEF-compliant.

If the tag turned out to be `NdefFormatable`, to put the `NdefMessage` on it, we first `connect()` to the tag, then `format()` it, supplying the message. `NdefFormatable` also supports `formatReadOnly()` for tags that support that mode — this will write the message on the tag, then block it from further updates. When we are done, we `close()` the connection.

If the tag turned out to be `Ndef` already, we `connect()` to it, then see if it is writable and has enough room. If it meets both of those criteria, we can emit the message via `writeNdefMessage()`, which overwrites the NDEF message that had already existed on the tag (if any). If the tag supported it, a call to `makeReadOnly()` would block further updates to the tag. Again, when we are done, we `close()` the connection.

All of the actual NFC I/O is performed in `doInBackground()`, because this I/O may take some time, and we do not want to block the main application thread while doing it.

Responding to a Tag

Writing to a tag is a bit complicated. Responding to an NDEF message on a tag is significantly easier.

If the foreground activity is not consuming NFC events — as `URLTagger` does in write mode — then Android will use normal `Intent` resolution with `startActivity()` to handle the tag. To respond to the tag, all you need to do is have an activity set up to watch for an `android.nfc.action.NDEF_DISCOVERED` `Intent`. To

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

get control ahead of the built-in Tags application, also have a `<data>` element that describes the sort of content or URL you are expecting to find on the tag.

For example, suppose you used the Android browser to visit some page on the CommonsWare Web site, and you wrote that to a tag using `URLTagger`. The URLTagger application has another activity, `URLHandler`, that will respond when you tap the newly-written tag from the home screen or anywhere else. It accomplishes this via a suitable `<intent-filter>`:

```
    <activity
      android:name="URLHandler"
      android:label="@string/app_name">
      <intent-filter android:label="@string/app_name">
        <action android:name="android.nfc.action.NDEF_DISCOVERED"/>

        <data
          android:host="commonsware.com"
          android:scheme="http"/>

        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
    </activity>
```

The `URLHandler` activity can then use `getIntent()` to retrieve the key pieces of data from the tag itself, if needed. In particular, the `EXTRA_NDEF_MESSAGES` `Parcelable` array extra will return an array of `NdefMessage` objects. Typically, there will only be one of these. You can call `getRecords()` on the `NdefMessage` to get at the array of `NdefRecord` objects (again, typically only one). Methods like `getPayload()` will allow you to get at the individual portions of the record.

The nice thing is that the URL still works, even if URLTagger is not on the device. In that case, the Tags application would react to the tag, and the user could tap on it to bring up a browser on this URL. A production application might create a Web page that tells the user about this great and wonderful app she can install, and provide links to the Play Store (or elsewhere) to go get the app.

Expected Pattern: Bootstrap

Tags tend to have limited capacity. Even in peer-to-peer settings, the effective bandwidth of NFC is paltry compared to anything outside of dial-up Internet access.

As a result, NFC will be used infrequently as the complete communications solution between a publisher and a device. Sometimes it will, when the content is specifically small, such as a contact (vCard) or event (iCalendar). But, for anything bigger than

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

that, NFC will serve more as a convenient bootstrap for more conventional communications options:

1. Embedding a URL in a tag, as the previous sample showed, allows an installed application to run or a Web site to be browsed
2. Embedding a Play Store URL in a tag allows for easy access to some specialized app (e.g., menu for a restaurant)
3. A multi-player game might use peer-to-peer NFC to allow local participants to rapidly connect into the same shared game area, where the game is played over the Internet or Bluetooth
4. And so on.

Mobile Devices are Mobile

Reading and writing NFC tags is a relatively slow process, mostly due to low bandwidth. It may take a second or two to actually complete the operation.

Users, however, are not known for their patience.

If a user moves their device out of range of the tag while Android is attempting to read it, Android simply will skip the dispatch. If, however, the tag leaves the signal area of the device while you are writing to it, you will get an `IOException`. At this point, the state of the tag is unknown.

You may wish to incorporate something into your UI to let the user know that you are working with the tag, encouraging them to leave the phone in place until you are done.

Enabled and Disabled

There are two separate system settings that control NFC behavior:

- The user could have NFC disabled outright, which you would detect by calling `isEnabled()` on your `NfcAdapter`
- The user could have NFC enabled but have Android Beam disabled, which you would detect by calling `isNdefPushEnabled()` on your `NfcAdapter`

As with most enabled/disabled settings, you cannot change these values yourself. On newer Android SDK versions, though, you can try to bring up the relevant

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Settings screens for the user to enable these features, by using the following activity action strings from the `android.provider.Settings` class:

- `ACTION_NFC_SETTINGS` for the main NFC settings screen (added in API Level 16)
- `ACTION_NFCSHARING_SETTINGS` for the Android Beam settings screen (added in API Level 14)

Android Beam

Android Beam is Google's moniker for peer-to-peer NFC messaging, with an emphasis — obviously — on Android apps. Rather than you tapping your NFC-capable Android device on a smart tag, you put it back-to-back with another NFC-capable Android device, and romance ensues.

Partially, this is simply one side of the exchange "pushing" an NDEF record, in a fashion that makes the other side of the exchange think that it is picking up a smart tag.

Partially, this is the concept of the "Android Application Record" (AAR), another NDEF record you can place in the NDEF message being pushed. This will identify the app you are trying to push the message to. If nothing on the device can handle the rest of the NDEF message, the AAR will lead Android to start up an app, or even lead the user to the Play Store to go download said app.

As the basis for explaining further how this all works, let's take a look at the `NFC/WebBeam` sample application. The UI consists of a `WebViewFragment`, in which we can browse to some Web page. Then, running this app on two NFC-capable devices, one app can "push" the URL of the currently-viewed Web page to the other app, which will respond by displaying that page. In this fashion, we are "sharing" a URL, without one side having to type it in by hand. And, while we are using this to share a URL, you could use Android Beam to share any sort of bootstrapping data, such as the user IDs of each person, for use in connecting to some common game server.

The Fragment

The fragment that implements our UI, `BeamFragment`, extends from `WebViewFragment`. In `onActivityCreated()`, we configure the `WebView`, load up Google's home page, and indicate that would like to participate in the action bar (via a call to `setHasOptionsMenu()`):

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@SuppressLint("SetJavaScriptEnabled")
@Override
public void onActivityCreated(Bundle savedInstanceState) {
  super.onActivityCreated(savedInstanceState);

  getWebView().setWebViewClient(new BeamClient());
  getWebView().getSettings().setJavaScriptEnabled(true);
  loadUrl("http://google.com");
  setHasOptionsMenu(true);
}
```

To keep all links within the `WebView`, we attached a `WebViewClient` implementation, named `BeamClient`, that just loads all requested URLs back into the `WebView`:

```
class BeamClient extends WebViewClient {
  @Override
  public boolean shouldOverrideUrlLoading(WebView wv, String url) {
    wv.loadUrl(url);

    return(true);
  }
}
```

We add one item to the action bar: a toolbar button (`R.id.beam`) that will be used to indicate we wish to beam the URL in our `WebView` to another copy of this application running on another NFC-capable Android device:

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  if (getContract().hasNFC()) {
    inflater.inflate(R.menu.actions, menu);
  }

  super.onCreateOptionsMenu(menu, inflater);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId() == R.id.beam) {
    getContract().enablePush();

    return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

So, when the app is initially launched, it will look something like this:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

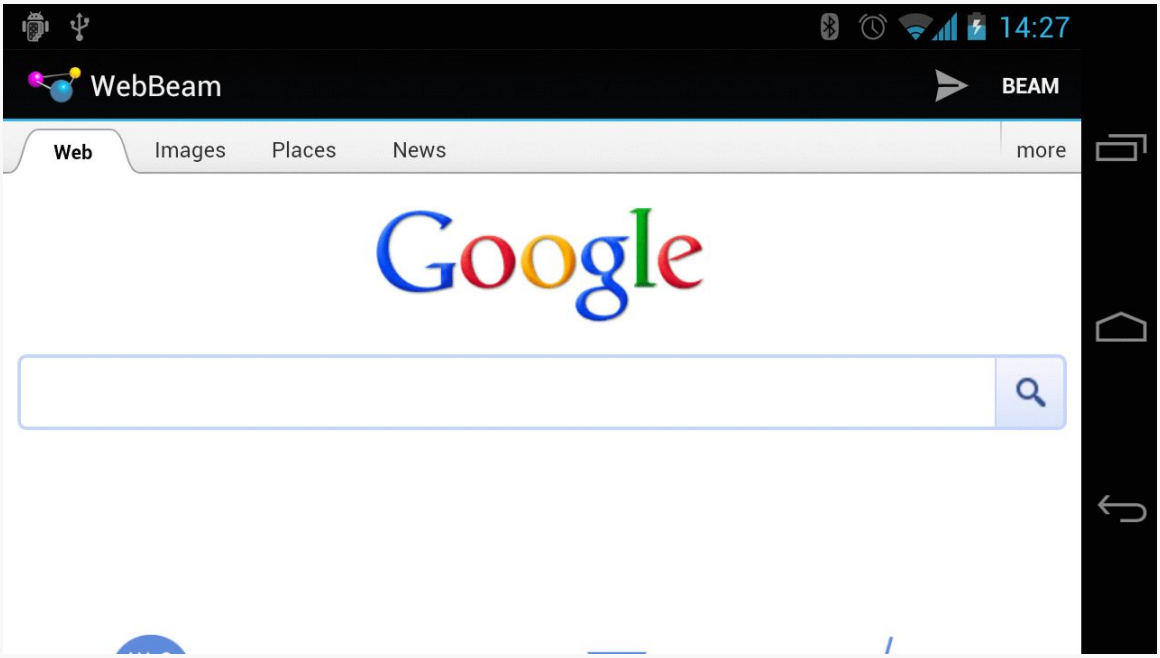


Figure 673: The WebBeam UI

The user can use Google to find a Web page worth beaming.

Requesting the Beam

Our hosting activity, `WebBeamActivity`, gets access to our `NfcAdapter`, as we did in the previous example:

```
    adapter=NfcAdapter.getDefaultAdapter(this);
```

When the user taps on our action bar item, the fragment calls `enablePush()` on the activity. `WebBeamActivity`, in turn, calls `setNdefPushMessageCallback()` on the `NfcAdapter`, supplying two parameters:

1. An implementation of the `NfcAdapter.CreateNdefMessageCallback` interface, used to let us know when another device is in range for us to beam to (in our case, `WebBeamActivity` implements this interface)
2. Our activity that is participating in this push

If something else comes to the foreground, `onStop()` will call a corresponding `disablePush()`, which also calls `setNdefPushMessageCallback()`, specifying a `null` first parameter, to turn off our request to beam:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  void enablePush() {
    adapter.setNdefPushMessageCallback(this, this);
  }

  void disablePush() {
    adapter.setNdefPushMessageCallback(null, this);
  }
```

In between the calls to `enablePush()` and `disablePush()`, if another NFC device comes in range that supports the NDEF push protocols, we're beamin'.

Sending the Beam

When our beam-enabled device encounters another beam-capable device, our `NfcAdapter.CreateNdefMessageCallback` is called with `createNdefMessage()`, where we need to prepare the `NfcMessage` to beam to the other party:

```
  @Override
  public NdefMessage createNdefMessage(NfcEvent arg0) {
    NdefRecord uriRecord=
        new NdefRecord(NdefRecord.TNF_MIME_MEDIA,
                       MIME_TYPE.getBytes(Charset.forName("US-ASCII")),
                       new byte[0],
                       beamFragment.getUrl()
                           .getBytes(Charset.forName("US-ASCII")));
    NdefMessage msg=
        new NdefMessage(
                        new NdefRecord[] {
                            uriRecord,

NdefRecord.createApplicationRecord("com.commonsware.android.webbeam") });

    return(msg);
  }
```

We first create a typical `NfcRecord`, in this case of `TNF_MIME_MEDIA`, with a MIME type defined in a static data member and payload consisting of the URL from our `WebView`:

```
  private static final String MIME_TYPE=
      "application/vnd.commonsware.sample.webbeam";
```

You might wonder why we are using `TNF_MIME_MEDIA`, instead of `TNF_WELL_KNOWN` and a subtype of `RTD_URI`, since our payload is a URL. The reason is that we need to have a unique MIME type for our message for the whole beam process to work properly, and `TNF_WELL_KNOWN` does not support MIME types. This is also why the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

MIME type is something distinctive, and not just `text/plain` — it has to be something only we will pick up.

Our `NfcMessage` then consists of *two* `NfcRecord` objects: the one we just created, and one created via the static `createApplicationRecord()` method on `NfcRecord`. This helper method creates an AAR record, identifying our application by its Android package name. This record must go last – Android will try to find an app to work with based on the other records first, before "failing over" to use the AAR.

Receiving the Beam

To receive our beam, our `WebBeamActivity` must be configured in the manifest to respond to `NDEF_DISCOVERED` actions with our unique MIME type:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.webbeam"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="14"/>

  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.NFC"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.Holo.Light.DarkActionBar">
    <activity
      android:name=".WebBeamActivity"
      android:label="@string/app_name"
      android:launchMode="singleTask"
      android:screenOrientation="landscape">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
      <intent-filter>
        <action android:name="android.nfc.action.NDEF_DISCOVERED"/>

        <category android:name="android.intent.category.DEFAULT"/>

        <data android:mimeType="application/vnd.commonsware.sample.webbeam"/>
      </intent-filter>
    </activity>
  </application>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
</manifest>
```

You will also notice that we set `android:launchMode="singleTask"` on this activity. That is so we will only have one instance of this activity, regardless of whether it is in the foreground or not. Otherwise, if we already have an instance of this activity, and we receive a beam, Android will create a *second* instance of this activity — when the user later presses BACK, they return to our first instance, and wonder why our app is broken.

If we receive the beam, we will get the `Intent` for the `NDEF_DISCOVERED` action *either* in `onCreate()` (if we were not already running) or `onNewIntent()` (if we were). In either case, we want to handle it the same way: pass the URL from the first record's payload to our `BeamFragment`. However, we cannot do that from `onCreate()` — the fragment will not have created the `WebView` yet. So, we use a trick: calling `post()` with a `Runnable` puts that `Runnable` on the *end* of the work queue for the main application thread. We can delay our processing of the `Intent` by this mechanism, so we can safely assume the `WebView` exists.

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    beamFragment=

(BeamFragment)getFragmentManager().findFragmentById(android.R.id.content);

    if (beamFragment == null) {
      beamFragment=new BeamFragment();

      getFragmentManager().beginTransaction()
                          .add(android.R.id.content, beamFragment)
                          .commit();
    }

    adapter=NfcAdapter.getDefaultAdapter(this);

    findViewById(android.R.id.content).post(new Runnable() {
      public void run() {
        handleIntent(getIntent());
      }
    });
  }

  @Override
  public void onNewIntent(Intent i) {
    handleIntent(i);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
private void handleIntent(Intent i) {
  if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(i.getAction())) {
    Parcelable[] rawMsgs=
        i.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
    NdefMessage msg=(NdefMessage)rawMsgs[0];
    String url=new String(msg.getRecords()[0].getPayload());

    beamFragment.loadUrl(url);
  }
}
```

The Scenarios

There are three possible scenarios, when we try beaming from one device to another:

1. The other device has our application installed, and it is running. In that case, our activity is brought to the foreground and the `Intent` is delivered to it, courtesy of our `NDEF_DISCOVERED` `<intent-filter>` with our unique MIME type.
2. The other device has our application installed, but it is not running. Android's `Intent` system handles this in the same general fashion as the first scenario, though it starts up a process for us and creates our activity instance anew in this case.
3. The other device does not have our application installed. Since nothing (hopefully) claims to support our unique MIME type, the AAR takes effect, and the user is led to the Play Store to go download our app (or, in this case, display an error message, as `WebBeam` is not in the Play Store).

Beaming Files

Android 4.1 (a.k.a., Jelly Bean) added in a far simpler facility for an app to beam a file to another device using the Android Beam system. You can use `setBeamPushUris()` or `setBeamPushUrisCallback()` on an `NfcAdapter` to hand Android one or more `Uri` objects representing files to be transferred. While the initial connection will be made via NFC and Android Beam, the actual data transfer will be via Bluetooth or WiFi, much more suitable than NFC for bulk data.

The difference between the two approaches is mostly when you provide the array of `Uri` objects. With `setBeamPushUris()`, you initiate the beam operation and supply the `Uri` values immediately. With `setBeamPushUrisCallback()`, you initiate the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

beam but do not supply the `Uri` values until the beam connection is established with the peer app.

The NFC/FileBeam sample application shows file-based beaming in action.

In our activity (`MainActivity`), in `onCreate()`, we check to make sure that Android Beam is enabled, via a call to `isNdefPushEnabled()` on our `NfcAdapter`. If it is, then we use `ACTION_GET_CONTENT` to retrieve some file from the user (MIME type wildcard of `*/*`):

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  adapter=NfcAdapter.getDefaultAdapter(this);

  if (!adapter.isNdefPushEnabled()) {
    Toast.makeText(this, R.string.sorry, Toast.LENGTH_LONG).show();
    finish();
  }
  else {
    Intent i=new Intent(Intent.ACTION_GET_CONTENT);

    i.setType("*/*");
    startActivityForResult(i, 0);
  }
}
```

In `onActivityResult()`, if we actually got a file (e.g., the result is `ACTION_OK`), we turn around and call `setBeamPushUris()` to pass that file to some peer device. We also set up a `Button` as our UI — clicking the `Button` will `finish()` the activity:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (requestCode==0 && resultCode==RESULT_OK) {
    adapter.setBeamPushUris(new Uri[] {data.getData()}, this);

    Button btn=new Button(this);

    btn.setText(R.string.over);
    btn.setOnClickListener(this);
    setContentView(btn);
  }
}
```

That is all there is to it. If you run this app and pick a file, then hold the device up to another Android 4.1+ device, you will be prompted to "Touch to Beam" — doing so will kick off the transfer. Once the transfer is shown on the receiving device, you can

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

pull the devices apart a bit, as the transfer will be proceeding over Bluetooth or WiFi. However, while Bluetooth ranges are much longer than NFC, you still need to keep the devices within a handful of meters of one another.

Note that the receiving device is not running our app. The OS handles the receipt of the transferred file, not our code. Similarly, the OS on the sending device is really the one responsible for the file transfer, so our app does not need the `INTERNET` or `BLUETOOTH` permissions. The downside is that we have no control over anything on the receiving side — the file is stored wherever the OS elects to put it, and the `Notification` it displays when complete will simply launch `ACTION_VIEW` on the pushed file.

Another Sample: SecretAgentMan

To provide another take on using these features of `NfcAdapter`, let's examine the [NFC/SecretAgentMan] sample application, originally written for a presentation at the 2012 droidcon UK conference. This combines writing to tags, directly beaming text to another device, and using `Uri`-based beaming, all in one app.

The UI of the app is a large `EditText` widget with an action bar:

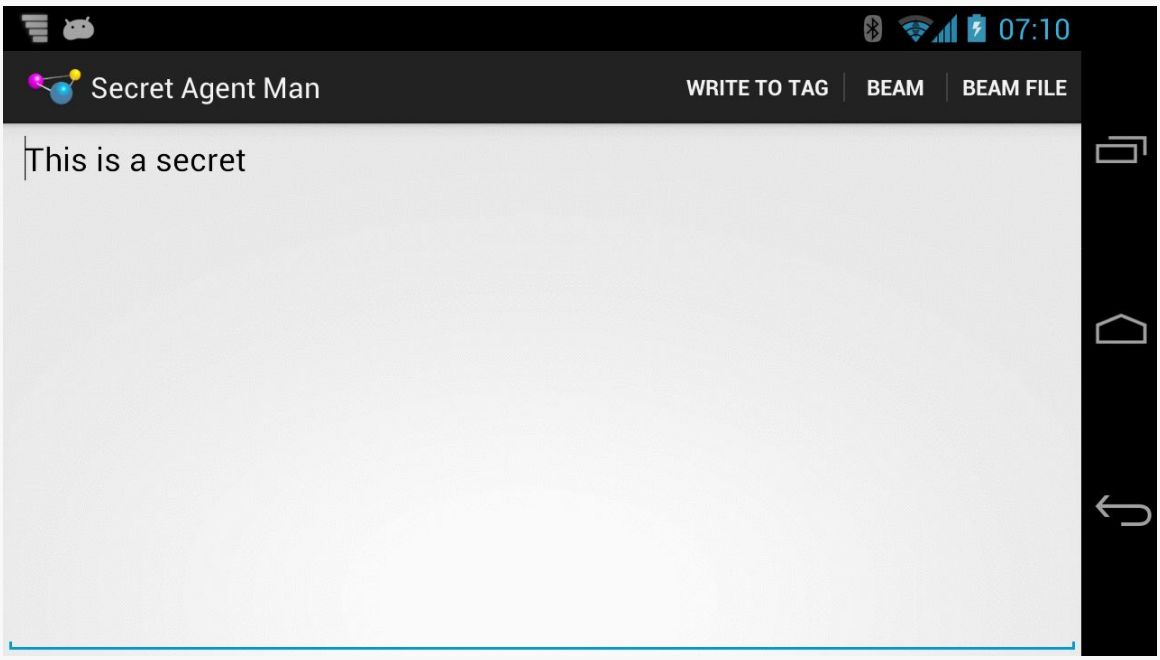


Figure 674: The SecretAgentMan UI

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

There are three action bar items, one each for the three operations: writing to a tag, directly beaming to another device, and beaming a file (represented via a `Uri`).

Configuration and Initialization

Our app is comprised of a single activity, named `MainActivity`. As part of our manifest setup, we request the `NFC` permission. And, since the app needs NFC to be useful, we also have a `<uses-feature>` element, stipulating that the device needs to have NFC, otherwise the app should not be shown in the Play Store:

```
<uses-permission android:name="android.permission.NFC"/>

<uses-feature
  android:name="android.hardware.nfc"
  android:required="true"/>
```

In `onCreate()` of `MainActivity`, we can then safely get access to an `NfcAdapter`, since the NFC hardware should exist and we have rights to use NFC:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  nfc=NfcAdapter.getDefaultAdapter(this);
  secretMessage=(EditText)findViewById(R.id.secretMessage);

  nfc.setOnNdefPushCompleteCallback(this, this);

  if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getIntent().getAction())) {
    readFromTag(getIntent());
  }
}
```

We also get our hands on the `EditText` widget, storing a reference to it in a data member named `secretMessage`. We will cover the rest of the initialization work in `onCreate()` later in this section, as we cover the code that needs that initialization.

Writing to the Tag

If the user chooses the "Write to Tag" action bar item, we call a `setUpWriteMode()` method from `onOptionsItemSelected()` of `MainActivity`. We maintain an `inWriteMode` `boolean` data member to track whether or not we are already trying to write to an NFC tag. If `inWriteMode` is false, we go ahead and take control over the NFC hardware to attempt to write to the next tag we see:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
void setUpWriteMode() {
  if (!inWriteMode) {
    IntentFilter discovery=
        new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
    IntentFilter[] tagFilters=new IntentFilter[] { discovery };
    Intent i=
        new Intent(this, getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP
            | Intent.FLAG_ACTIVITY_CLEAR_TOP);
    PendingIntent pi=PendingIntent.getActivity(this, 0, i, 0);

    inWriteMode=true;
    nfc.enableForegroundDispatch(this, pi, tagFilters, null);
  }
}
```

To do that, we:

- Create an `IntentFilter` for `ACTION_TAG_DISCOVERED`
- Create a `PendingIntent` for an `Intent` pointing back to this same activity instance (using `getClass()` to identify the instance, plus `FLAG_ACTIVITY_SINGLE_TOP` and `FLAG_ACTIVITY_CLEAR_TOP` to route control back to our running instance)
- Call `enableForegroundDispatch()` on our `NfcAdapter`, to route newly-discovered tags to us, with the `IntentFilter` identifying the tag-related events we are interested in, and the `PendingIntent` identifying what to do when such a tag is encountered

Once our activity is finishing (e.g., the user presses BACK), we need to clean up our write-to-tag logic. This is kicked off in `onPause()` of `MainActivity`:

```
@Override
public void onPause() {
  if (isFinishing()) {
    cleanUpWritingToTag();
  }

  super.onPause();
```

All we do in `cleanUpWritingToTag()` is discontinue our foreground control over the NFC hardware:

```
void cleanUpWritingToTag() {
  nfc.disableForegroundDispatch(this);
  inWriteMode=false;
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If, before that occurs, the device is tapped on a tag, our activity should regain control in `onNewIntent()` as a result of our `PendingIntent` having been executed:

```
@Override
protected void onNewIntent(Intent i) {
  if (inWriteMode
      && NfcAdapter.ACTION_TAG_DISCOVERED.equals(i.getAction())) {
    writeToTag(i);
  }
  else if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(i.getAction())) {
    readFromTag(i);
  }
}
```

If we are in write mode, and if the `Intent` that was just used with `startActivity()` was `ACTION_TAG_DISCOVERED`, we call our `writeToTag()` method to actually start writing information to the tag:

```
void writeToTag(Intent i) {
  Tag tag=i.getParcelableExtra(NfcAdapter.EXTRA_TAG);
  NdefMessage msg=
      new NdefMessage(new NdefRecord[] { buildNdefRecord() });

  new WriteTagTask(this, msg, tag).execute();
}
```

To write to the tag, we get our `Tag` out of its `Intent` extra (keyed by `EXTRA_TAG`). Then, we build an `NfcMessage` to write to the tag, getting its `NfcRecord` from `buildNdefRecord()`:

```
NdefRecord buildNdefRecord() {
  return(new NdefRecord(NdefRecord.TNF_MIME_MEDIA,
                        MIME_TYPE.getBytes(), new byte[] {},
                        secretMessage.getText().toString().getBytes()));
}
```

Our NDEF record will be of a specific MIME type, represented by a static data member named `MIME_TYPE`:

```
private static final String MIME_TYPE="vnd.secret/agent.man";
```

The payload of the NDEF record is our "secret message" from the `secretMessage` `EditText` widget.

The `writeToTag()` method then kicks off the same `WriteTagTask` that we used earlier in this chapter:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.jimmyb;

import android.nfc.NdefMessage;
import android.nfc.Tag;
import android.nfc.tech.Ndef;
import android.nfc.tech.NdefFormatable;
import android.os.AsyncTask;
import android.util.Log;
import android.widget.Toast;

class WriteTagTask extends AsyncTask<Void, Void, Void> {
  MainActivity host=null;
  NdefMessage msg=null;
  Tag tag=null;
  String text=null;

  WriteTagTask(MainActivity host, NdefMessage msg, Tag tag) {
    this.host=host;
    this.msg=msg;
    this.tag=tag;
  }

  @Override
  protected Void doInBackground(Void... arg0) {
    int size=msg.toByteArray().length;

    try {
      Ndef ndef=Ndef.get(tag);

      if (ndef == null) {
        NdefFormatable formatable=NdefFormatable.get(tag);

        if (formatable != null) {
          try {
            formatable.connect();

            try {
              formatable.format(msg);
            }
            catch (Exception e) {
              text=host.getString(R.string.tag_refused_to_format);
            }
          }
          catch (Exception e) {
            text=host.getString(R.string.tag_refused_to_connect);
          }
          finally {
            formatable.close();
          }
        }
        else {
          text=host.getString(R.string.tag_does_not_support_ndef);
        }
      }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      else {
        ndef.connect();

        try {
          if (!ndef.isWritable()) {
            text=host.getString(R.string.tag_is_read_only);
          }
          else if (ndef.getMaxSize() < size) {
            text=host.getString(R.string.message_is_too_big_for_tag);
          }
          else {
            ndef.writeNdefMessage(msg);
            text=host.getString(R.string.success);
          }
        }
        catch (Exception e) {
          text=host.getString(R.string.tag_refused_to_connect);
        }
        finally {
          ndef.close();
        }
      }
    }
    catch (Exception e) {
      Log.e("URLTagger", "Exception when writing tag", e);
      text=host.getString(R.string.general_exception) + e.getMessage();
    }

    return(null);
  }

  @Override
  protected void onPostExecute(Void unused) {
    host.cleanUpWritingToTag();

    if (text != null) {
      Toast.makeText(host, text, Toast.LENGTH_SHORT).show();
    }
  }
}
```

The net result is that if the user taps the “Write to Tag” action bar item, then taps and holds the device to a tag, we will write a message to the tag and display a `Toast` when we are done.

And, yes, this is a surprising amount of code for what really should be a simple operation...

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Reading from the Tag

We can set up `MainActivity` to respond to tags similar to the one we wrote — ones that have the desired MIME Type — via an `android.nfc.action.NDEF_DISCOVERED` `<intent-filter>`:

```
      <intent-filter android:label="@string/app_name">
        <action android:name="android.nfc.action.NDEF_DISCOVERED"/>

        <data android:mimeType="vnd.secret/agent.man"/>

        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
```

In both `onCreate()` and `onNewIntent()`, if the `Intent` that started our activity is an `NDEF_DISCOVERED` `Intent`, we route control to a `readFromTag()` method:

```
  void readFromTag(Intent i) {
    Parcelable[] msgs=
        (Parcelable[])i.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);

    if (msgs.length > 0) {
      NdefMessage msg=(NdefMessage)msgs[0];

      if (msg.getRecords().length > 0) {
        NdefRecord rec=msg.getRecords()[0];

        secretMessage.setText(new String(rec.getPayload(), US_ASCII));
      }
    }
  }
```

In principle, there could be several NDEF messages on the tag, but we only pay attention to the first element, if any, of the `EXTRA_NDEF_MESSAGES` array of `Parcelable` objects on the `Intent`. Similarly, in principle, there could be several NDEF records in the first message, but we only examine the first element out of the array of `NdefRecord` objects contained in the `NdefMessage`. From there, we extract our secret message and display it by means of putting it in the `EditText` widget.

Beaming the Text

This sample only supports beaming — whether of NDEF messages directly or of a file — if we are on API Level 16 or higher. Hence, in `onCreateOptionsMenu()`, we check our version and only enable our default-disabled beam action bar items if:

- We are on API Level 16 or higher, and

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- NDEF push mode is enabled, via a call to `isNdefPushEnabled()` on our `NfcAdapter`:

```
  @TargetApi(16)
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.activity_main, menu);

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN) {
      menu.findItem(R.id.simple_beam)
          .setEnabled(nfc.isNdefPushEnabled());
      menu.findItem(R.id.file_beam).setEnabled(nfc.isNdefPushEnabled());
    }

    return(super.onCreateOptionsMenu(menu));
  }
```

If the user taps on the "Beam" action bar item, we call an `enablePush()` method from `onOptionsItemSelected()`, which simply enables push mode:

```
  void enablePush() {
    nfc.setNdefPushMessageCallback(this, this);
  }
```

We arrange for the activity itself to be the `CreateNdefMessageCallback` necessary for push mode. That requires us to implement `createNdefMessage()`, which will be called if we are in push mode and a push-compliant device comes within range:

```
  @Override
  public NdefMessage createNdefMessage(NfcEvent event) {
    return(new NdefMessage(
                            new NdefRecord[] {
                                buildNdefRecord(),

NdefRecord.createApplicationRecord("com.commonsware.android.jimmyb") }));
  }
```

Here, we create an `NdefMessage` similar to the one we wrote to the tag earlier in this sample. However, we also attach an Android Application Record (AAR), by means of the static `createApplicationRecord()` method on `NdefRecord`. This, in theory, will help route the push to our app on the other device, including downloading it from the Play Store if needed (and, of course, if it actually existed on the Play Store, which it does not).

Back up in `onCreate()`, we call `setOnNdefPushCompleteCallback()`, to be notified of when a push operation is completed. Once again, we set up `MainActivity` to be the callback, this time by implementing the `OnNdefPushCompleteCallback` interface.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That, in turn, requires us to implement `onNdefPushComplete()`, where we disable push mode via a call to `setNdefPushMessageCallback()` with a `null` listener:

```
@Override
public void onNdefPushComplete(NfcEvent event) {
  nfc.setNdefPushMessageCallback(null, this);
}
```

To receive the beam, we only need our existing logic to read from the tag, as on the receiving side, a push is indistinguishable from reading a tag, and we are using the same MIME type for both the message written to the tag and the message we are pushing.

Beaming the File

If the user taps the "Beam File" action bar item, we find some file to beam, by means of an `ACTION_GET_CONTENT` request and `startActivityForResult()`:

```
    case R.id.file_beam:
      Intent i=new Intent(Intent.ACTION_GET_CONTENT);

      i.setType("*/*");
      startActivityForResult(i, 0);
      return(true);
```

In `onActivityResult()`, if the request succeeded, we use `setBeamPushUris()` to tell Android to beam the selected file to another device. Nothing more is needed on our side, and the receipt of the file is handled entirely by the OS, not our application code, so there is nothing to be written for that.

This code assumes the NFC adapter is enabled. We could check that via a call to `isEnabled()` on our `NfcAdapter`. If it is not enabled, we could — on user request — bring up the Settings activity for configuring NFC, via `startActivity(new Intent(Settings.ACTION_NFC_SETTINGS))`. However, oddly, this `Intent` action is only available on Android 4.1 (API Level 16) and higher, despite NFC having been available for some time previously.

This code ignores the possibility of doing the simple beam (not the file-based beam) on Android 4.0.x devices. That is because the `isNdefPushEnabled()` method was not added until Android 4.1, and therefore we do not know whether or not we can actually do a beam.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If `isNdefPushEnabled()` returns `false`, we simply disable some action bar items. Alternatively, we could use `startActivity(new Intent(Settings.ACTION_NFCSHARING_SETTINGS))`, on API Level 14 and higher, to bring up the beam screen in Settings, to allow the user to toggle beam support on.

Additional Resources

To help make sense of the tags that you are trying to use with your app, you may wish to grab the NFC TagInfo application off of the Google Play Store. This application simply scans a tag and allows you to peruse all the details of that tag, including the supported technologies (e.g., does it support NDEF? is it `NdefFormatable`?), the NDEF records, and so on.

To learn more about NFC on Android — beyond this chapter or the Android developer documentation – this Google I|O 2011 presentation is recommended.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Device Administration

Balding authors of Android books often point out that enterprises and malware authors have the same interests: they want to take control of a device away from the person that is holding it and give that control to some other party. Android, being a consumer operating system, is designed to defend against malware, and so enterprises can run into issues.

However, Android does have a growing area of device administration APIs, that allow carefully-constructed and installed applications to exert some degree of control over the device, how it is configured, and how it operates.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on broadcast `Intents`.

Objectives and Scope

One might read the phrase "device administration" and assume that somebody, using these APIs, could do anything they want on the device.

That's not quite what "device administration" means in this case.

Rather, the device administration APIs serve three main roles:

1. They allow an application to dictate how well a device is secured, from the password required in the OS lock screen to whether the device should have full-disk encryption

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. They allow an application to find out when security issues might arise, notably failed password attempts
3. They allow an application to lock the device, disable its cameras, or even perform a "wipe" (i.e., factory reset)

The user, however, has to agree to enable a device administration app. It does not magically get all these powers simply by being installed. What the user gets from agreeing to this is access to something that otherwise would be denied (e.g., to use Enterprise App X, you must agree to allow it to be a device administrator).

Defining and Registering an Admin Component

There are four pieces for defining and registering a device administration app: creating the metadata, adding the `<receiver>` to the manifest, implementing that `BroadcastReceiver`, and telling Android to ask the user to agree to allow the app to a device administrator.

Here, we will take a peek at the `DeviceAdmin/LockMeNow` sample application.

The Feature

Apps implementing device administrators should add a `<uses-feature>` element with a name of `android.software.device_admin`, indicating whether or not they require this device feature to exist. This can be used by the Play Store to filter your app from being available on devices that, for one reason or another, do not offer this capability.

The Metadata

As with app widgets and other Android facilities, you will need to define a metadata file as an XML resource, describing in greater detail what your device administration app wishes to do. This information will determine what you will be allowed to do once the user approves your app, and what you list here will be displayed to the user when you request such approval.

The `DeviceAdminInfo` class has a series of static data members (e.g., `USES_ENCRYPTED_STORAGE`) that represent specific policies that your device administrator app could use. The documentation for each of those static data members lists the corresponding element that goes in this XML metadata file (e.g., `<encrypted-storage>`). These elements are wrapped in a `<uses-policies>` element,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

which itself is wrapped in a `<device-admin>` element. The range of possible policies is shown in the following sample XML metadata file:

```
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">
  <uses-policies>
    <disable-camera />
    <encrypted-storage />
    <expire-password />
    <force-lock />
    <limit-password />
    <reset-password />
    <watch-login />
    <wipe-data />
  </uses-policies>
</device-admin>
```

Here, we:

- Intend to disable the cameras, if needed
- Will ask the user to encrypt their device storage, if it has not been done already
- Will set an expiration time for the user's password, after which they will need to set up a new one
- Intend to lock the device, if needed
- Will set criteria for password quality, such as minimum length
- Intend to forcibly reset the user's password, if needed
- Intend to monitor for failed and successful login attempts
- Intend to wipe the device, if needed

Choose which of those policies you need — the fewer you request, the more likely it is the user will not wonder about your intentions. In your project's `res/xml/` directory, create a file that looks like the above with the policies you wish. You can name this file whatever you want (e.g., `device_admin.xml`), within standard Android resource naming rules.

The Manifest

In the manifest, you will need to declare a `<receiver>` element for the `DeviceAdminReceiver` component that you will write. This component not only is the embodiment of the device admin capabilities of your app, but it will be the one notified of failed logins and other events.

For example, here is the `<receiver>` element from the `LockMeNow` sample app:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    <receiver
      android:name="AdminReceiver"
      android:permission="android.permission.BIND_DEVICE_ADMIN">
      <meta-data
        android:name="android.app.device_admin"
        android:resource="@xml/device_admin"/>

      <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED"/>
      </intent-filter>
    </receiver>
```

There are three things distinctive about this element compared to your usual `<receiver>` element:

1. It requires that whoever sends broadcasts to it hold the `BIND_DEVICE_ADMIN` permission. Since that permission is protected and can only be held by apps signed with the firmware's signing key, you can be reasonably assured that any events sent to you are real.
2. It has the `<meta-data>` child element pointing to our device administration metadata from the previous section.
3. It registers for `android.app.action.DEVICE_ADMIN_ENABLED` broadcasts via its `<intent-filter>` — this is the broadcast that will be used to notify you about failed logins or other events.

The Receiver

The `DeviceAdminReceiver` itself needs to exist as a component in your app, registered in the manifest as shown above. At minimum, though, it does not need to override any methods, such as the implementation from the `LockMeNow` sample app:

```
package com.commonsware.android.lockme;

import android.app.admin.DeviceAdminReceiver;

public class AdminReceiver extends DeviceAdminReceiver {
}
```

By requesting the `DEVICE_ADMIN_ENABLED` broadcasts, we could get control when we are enabled by overriding an `onEnabled()` method. We could also register for other broadcasts (e.g., `ACTION_PASSWORD_FAILED`) and implement the corresponding callback method on our `DeviceAdminReceiver` (e.g., `onPasswordFailed()`).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Demand for Device Domination

Simply having this component in our manifest, though, is insufficient. The user must proactively agree to allow us to administer their device. And, since this is potentially very dangerous, a simple permission was deemed to also be insufficient. Instead, we need to ask the user to approve us as a device administrator from our app, typically from an activity.

In the case of `LockMeNow`, the UI is just a really big button, tied to a `lockMeNow()` method on our `LockMeNowActivity`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <Button
    android:id="@+id/Button1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:onClick="lockMeNow"
    android:text="@string/lock_me"
    android:textColor="#FFFF0000"
    android:textSize="40sp"
    android:textStyle="bold"/>

</LinearLayout>
```

In `onCreate()` of the activity, in addition to loading up the UI via `setContentView()`, we create a `ComponentName` object identifying our `AdminReceiver` component. We also request access to the `DevicePolicyManager`, via a call to `getSystemService()`. `DevicePolicyManager` is our gateway for making direct requests for device administration operations, such as locking the device:

```
package com.commonsware.android.lockme;

import android.app.Activity;
import android.app.admin.DevicePolicyManager;
import android.content.ComponentName;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class LockMeNowActivity extends Activity {
  private DevicePolicyManager mgr=null;
  private ComponentName cn=null;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);
    cn=new ComponentName(this, AdminReceiver.class);
    mgr=(DevicePolicyManager)getSystemService(DEVICE_POLICY_SERVICE);
  }

  public void lockMeNow(View v) {
    if (mgr.isAdminActive(cn)) {
      mgr.lockNow();
    }
    else {
      Intent intent=
          new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
      intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, cn);
      intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION,
                      getString(R.string.device_admin_explanation));
      startActivity(intent);
    }
  }
}
```

In `lockMeNow()`, we ask the `DevicePolicyManager` if we have already been registered as a device administrator, by calling `isAdminActive()`, supplying the `ComponentName` of our `DeviceAdminReceiver` that should be so registered. If that returns `false`, then the user has not approved us as a device administrator yet, so we need to ask them to do so. To do that, you:

- Create an `Intent` for the `DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN` action
- Add the `ComponentName` of our `DeviceAdminReceiver` as an extra, keyed as `DevicePolicyManager.EXTRA_DEVICE_ADMIN`
- Add another extra, `DevicePolicyManager.EXTRA_ADD_EXPLANATION`, which is some text to show the user as part of the authorization screen, to explain why we need to be a device admin
- Start up an activity using that `Intent`, via `startActivity()`

If you run this on a device, then tap the button, the first time you do so the user will be prompted to agree to making the app be a device administrator:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

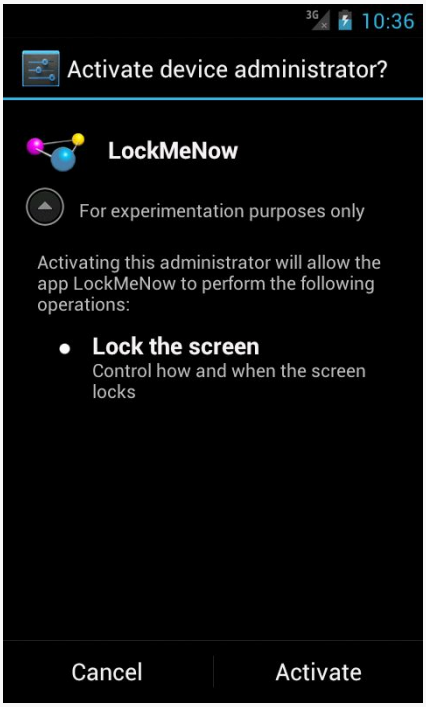


Figure 675: The Activate Device Administrator Screen

The "For experimentation purposes only" is the value of our `DevicePolicyManager.EXTRA_ADD_EXPLANATION` extra, loaded from a string resource.

If the user clicks "Activate", and you overrode `onEnabled()` in your `DeviceAdminReceiver`, that will be called to let you know that you have been approved and can perform device administration functions. Your component will also appear in the list of device administrators in the Settings app:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

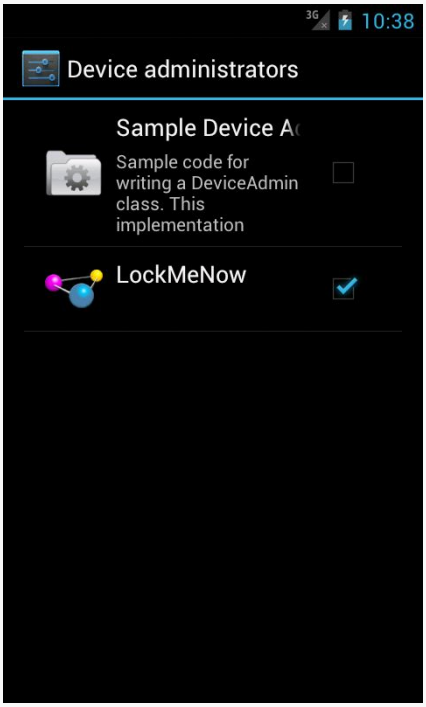


Figure 676: The Device Administrator List

The user can, at any time, uncheck you in this list and disable you. You can find out about this by having your `DeviceAdminReceiver` listen for `ACTION_DEVICE_ADMIN_DISABLE_REQUESTED` broadcasts and overriding the `onDisableRequested()` method, where you can return the text of a message to be displayed to the user confirming that they do indeed wish to go ahead with the disable operation. To find out if they go through with it, your `DeviceAdminReceiver` can listen for `ACTION_DEVICE_ADMIN_DISABLED` broadcasts and override `onDisabled()`.

Going Into Lockdown

Given that the user has approved your device administration request, and given that you requested `<force-lock>` in your metadata, you can call `lockNow()` on a `DevicePolicyManager`. That will immediately lock the device and (generally) turn off the screen. It is as if the user pressed the POWER button on the device.

The `LockItNow` sample app does this if, when the user clicks the really big button, it detects that it is already a device administrator. If you test this on a device, it will

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

behave as though the user pressed POWER; on an emulator, you will need to press the HOME button to "power on" the screen and be able to re-enter your emulator.

You can also call:

- `setCameraDisabled()` to disable all cameras, if you requested `<disable-camera>` in the metadata. Note that this disables all cameras; there is no provision at this time to disable individual cameras separately.
- `wipeData()`, which performs what amounts to a factory reset — it leaves external storage alone but wipes the contents of internal storage as part of a reboot. This requires the `<wipe-data>` policy in the metadata.
- `setKeyguardDisabledFeatures()`, to control whether or not the lockscreen allows direct access to the camera and/or app widgets (lockscreen app widgets are described in the chapter on app widgets)

For example, the latter feature, while available in the Android SDK, is not built into the Settings app of Android 4.2. As a result, users need a third-party app to toggle on or off lockscreen access to the camera and app widgets. One such third-party app is LockscreenLocker, released as open source by the author of this book.

Basically, the app presents you with two `Switch` widgets to control the camera and app widgets on the lock screen. First, though, it shows you a message and a `Button`, if the app is not set up as a device administrator:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

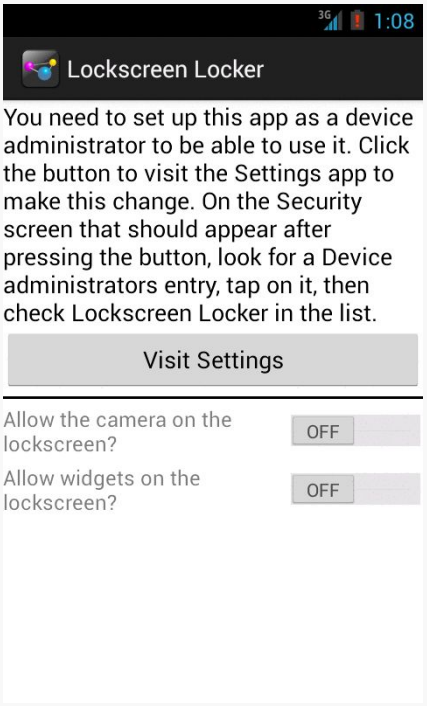


Figure 677: LockscreenLocker, On Initial Run

Once that is complete, the `Switch` widgets become enabled and usable:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

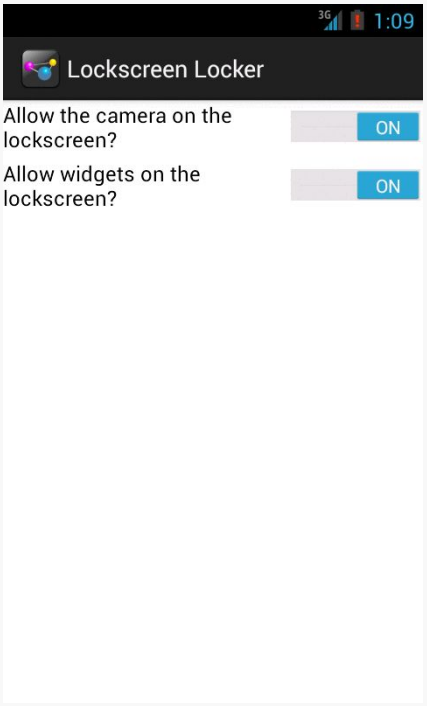


Figure 678: LockscreenLocker, After Being Made a Device Admin

The device admin metadata for this app specifies that we want to control keyguard features:

```
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-policies>
    <disable-keyguard-features/>
  </uses-policies>

</device-admin>
```

Note that, at the time of this writing, there is a flaw in the Android developer documentation — the correct element to have in the metadata is `<disable-keyguard-features/>`, not `<disable-keyguard-widgets>`. You can track this issue to see when this documentation bug has been repaired.

Our device admin component, `LockscreenAdminReceiver`, is empty, because there are no events that we are trying to listen to:

```
public class LockscreenAdminReceiver extends DeviceAdminReceiver {
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, we still need the `LockscreenAdminReceiver`, as it is the component that is tied to our device admin metadata and indicates to the system that we should be an option in Settings for available device administrators.

Our activity layout contains all the requisite widgets: a `TextView` for the message, a `Button` to jump to the Settings app, a `View` to serve as a divider, and a pair of `Switch` widgets to manage the lockscreen settings:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <TextView
    android:id="@+id/setupMessage"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/setup_message"
    android:textAppearance="?android:attr/textAppearanceMedium"
    android:visibility="gone"/>

  <Button
    android:id="@+id/setup"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="showSettings"
    android:text="@string/visit_settings"
    android:visibility="gone"/>

  <View
    android:id="@+id/divider"
    android:layout_width="match_parent"
    android:layout_height="2dip"
    android:layout_marginBottom="4dip"
    android:layout_marginTop="4dip"
    android:background="#FF000000"
    android:visibility="gone"/>

  <Switch
    android:id="@+id/camera"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/allow_camera"/>

  <Switch
    android:id="@+id/widgets"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="4dip"
    android:text="@string/allow_widgets"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
</LinearLayout>
```

In onCreate() of our activity (MainActivity), we request a DevicePolicyManager, set up a ComponentName identifying our DeviceAdminReceiver implementation (LockscreenAdminReceiver), and hook up the activity to know about changes in the state of the Switch widgets:

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    mgr=(DevicePolicyManager)getSystemService(DEVICE_POLICY_SERVICE);
    cn=new ComponentName(this, LockscreenAdminReceiver.class);

    camera=(CompoundButton)findViewById(R.id.camera);
    camera.setOnCheckedChangeListener(this);

    widgets=(CompoundButton)findViewById(R.id.widgets);
    widgets.setOnCheckedChangeListener(this);
  }
```

In onResume(), we check to see if our DeviceAdminReceiver is active — in other words, whether the user has set us up as being a device administrator or not:

```
  @Override
  public void onResume() {
    super.onResume();

    if (mgr.isAdminActive(cn)) {
      toggleWidgets(true);

      int status=mgr.getKeyguardDisabledFeatures(cn);

      camera.setChecked(!((status &
DevicePolicyManager.KEYGUARD_DISABLE_SECURE_CAMERA) ==
DevicePolicyManager.KEYGUARD_DISABLE_SECURE_CAMERA));
      widgets.setChecked(!((status &
DevicePolicyManager.KEYGUARD_DISABLE_WIDGETS_ALL) ==
DevicePolicyManager.KEYGUARD_DISABLE_WIDGETS_ALL));
    }
    else {
      toggleWidgets(false);
    }
  }
```

We toggle the visibility and enabled settings of our widgets based upon whether we are a device administrator or not, in a toggleWidgets() private method:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private void toggleWidgets(boolean enable) {
    int visibility=(enable ? View.GONE : View.VISIBLE);

    camera.setEnabled(enable);
    widgets.setEnabled(enable);

    findViewById(R.id.divider).setVisibility(visibility);
    findViewById(R.id.setup).setVisibility(visibility);
    findViewById(R.id.setupMessage).setVisibility(visibility);
  }
```

`onResume()` also sets the state of our `Switch` widgets based upon the current state of the keyguard features, by calling `getKeyguardDisabledFeatures()` on the `DevicePolicyManager`. This returns a bit set of which features are disabled, with `DevicePolicyManager.KEYGUARD_DISABLE_SECURE_CAMERA` and/or `DevicePolicyManager.KEYGUARD_DISABLE_WIDGETS_ALL` possibly being set.

At the outset, after being installed, we will not be a device administrator, so the `Switch` widgets will be disabled and the `Button` will be visible. We simply send the user to the security screen in the Settings app if they click that button:

```
  public void showSettings(View v) {
    startActivity(new Intent(Settings.ACTION_SECURITY_SETTINGS));
  }
```

When the user toggles a `Switch`, our activity will be called with `onCheckedChanged()`. There, we need to call `setKeyguardDisabledFeatures()` with a new bit set, toggling on or off a bit based on the user's chosen values in the UI:

```
  @Override
  public void onCheckedChanged(CompoundButton buttonView,
                               boolean isChecked) {
    int status=mgr.getKeyguardDisabledFeatures(cn);

    if (buttonView == camera) {
      if (isChecked) {
        mgr.setKeyguardDisabledFeatures(cn, status
            & ~DevicePolicyManager.KEYGUARD_DISABLE_SECURE_CAMERA);
      }
      else {
        mgr.setKeyguardDisabledFeatures(cn, status
            | DevicePolicyManager.KEYGUARD_DISABLE_SECURE_CAMERA);
      }
    }
    else {
      if (isChecked) {
        mgr.setKeyguardDisabledFeatures(cn, status
            & ~DevicePolicyManager.KEYGUARD_DISABLE_WIDGETS_ALL);
      }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      else {
        mgr.setKeyguardDisabledFeatures(cn, status
            | DevicePolicyManager.KEYGUARD_DISABLE_WIDGETS_ALL);
      }
    }
  }
```

Note that we have the `Switch` widgets set up for positive statements (e.g., "enable the camera"), while the bit set uses negative statements (e.g., "disable the camera"). That makes toggling the bit set a "bit" more complicated, to ensure that we are applying the user's choices correctly.

Passwords and Device Administration

One popular facet of the device administration APIs is for an app to mandate a certain degree of password quality. The app might then fail to operate if the current password does not meet the requested quality standard.

Mandating Quality of Security

You can call various setters on `DevicePolicyManager` to dictate your minimum requirements for the password that the user uses to get past the lock screen. Examples include:

- `setPasswordMinimumLength()`
- `setPasswordQuality()` (with an integer flag describing the type of "quality" you seek, such as `PASSWORD_QUALITY_NUMERIC` if a PIN is OK, or `PASSWORD_QUALITY_COMPLEX` if you require mixed case and numbers and such)
- `setPasswordMinimumLowerCase()` (indicating how many lowercase letters are required at minimum in the user's password)

All of these require the `<limit-password>` policy be requested in the metadata.

Then, you can call `isActivePasswordSufficient()` to determine if the current password meets your requirements. If it does not, you might elect to disable certain functionality. Or, if you requested the `<reset-password>` policy in the metadata, you can call `resetPassword()` to force the user to come up with a password meeting your requirements.

Similarly, you can also call `getStorageEncryptionStatus()` on `DevicePolicyManager` to find out whether full-disk encryption is active, inactive, or

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

unavailable on this particular device. If it is inactive, and you requested the `<encrypted-storage>` policy in your metadata, you can call `setStorageEncryption()` to demand it, and start the encryption process via starting the `ACTION_START_ENCRYPTION` activity.

Establishing Password Requirements

To see password quality enforcement in action, let us examine the DeviceAdmin/PasswordEnforcer sample application.

The activity (`MainActivity`) is fairly short, and much of its code is based off of the earlier `LockMeNow` sample:

```
package com.commonsware.android.pwenforce;

import android.app.Activity;
import android.app.admin.DevicePolicyManager;
import android.content.ComponentName;
import android.content.Intent;
import android.os.Bundle;
import android.widget.Toast;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    ComponentName cn=new ComponentName(this, AdminReceiver.class);
    DevicePolicyManager mgr=
        (DevicePolicyManager)getSystemService(DEVICE_POLICY_SERVICE);

    if (mgr.isAdminActive(cn)) {
      int msgId;

      if (mgr.isActivePasswordSufficient()) {
        msgId=R.string.compliant;
      }
      else {
        msgId=R.string.not_compliant;
      }

      Toast.makeText(this, msgId, Toast.LENGTH_LONG).show();
    }
    else {
      Intent intent=
          new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
      intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, cn);
      intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION,
                      getString(R.string.device_admin_explanation));
      startActivity(intent);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }

    finish();
  }
}
```

In `onCreate()`, after obtaining a `DevicePolicyManager`, we see if our app has been designated by the user as a device administrator. If not — which will be the case when the app is first installed — we use an `ACTION_ADD_DEVICE_ADMIN` `Intent` and `startActivity()` to steer the user towards making our app be a device administrator.

If the user does make our app be a device administrator, our `AdminReceiver` will get control in `onEnabled()`, as we have registered it for `DEVICE_ADMIN_ENABLED` broadcasts in the manifest. In `onEnabled()`, we mandate that the password for the device must be alphanumeric, via a call to `setPasswordQuality()` on the `DevicePolicyManager`:

```
  @Override
  public void onEnabled(Context ctxt, Intent intent) {
    ComponentName cn=new ComponentName(ctxt, AdminReceiver.class);
    DevicePolicyManager mgr=

(DevicePolicyManager)ctxt.getSystemService(Context.DEVICE_POLICY_SERVICE);

    mgr.setPasswordQuality(cn,
                           DevicePolicyManager.PASSWORD_QUALITY_ALPHANUMERIC);

    onPasswordChanged(ctxt, intent);
  }
```

We will see the role of the `onPasswordChanged()` method, called late in `onEnabled()`, later in this chapter.

Back in `onCreate()` of our `MainActivity`, if we *are* a device administrator, then we know that the `setPasswordQuality()` call has been made, and so we can check to see if the current password meets our standards via a call to `isActivePasswordSufficient()` on the `DevicePolicyManager`. The app displays a `Toast` showing whether the password is or is not currently "sufficient".

Password-Related Events

Via appropriate actions in our `<intent-filter>` for our `DeviceAdminReceiver`, and associated callback methods, we can find out other things that go on with respect to the password:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- ACTION_PASSWORD_CHANGED informs us when the user has changed her password
- ACTION_PASSWORD_FAILED informs us when somebody tries to enter a password, and the password was incorrect
- ACTION_PASSWORD_SUCCEEDED informs us when the user has successfully entered the password and unlocked the device... after an attempt had previously failed

The PasswordEnforcer sample registers for all of these in the manifest:

```
    <receiver
      android:name="AdminReceiver"
      android:permission="android.permission.BIND_DEVICE_ADMIN">
      <meta-data
        android:name="android.app.device_admin"
        android:resource="@xml/device_admin"/>

      <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED"/>
        <action android:name="android.app.action.ACTION_PASSWORD_CHANGED"/>
        <action android:name="android.app.action.ACTION_PASSWORD_FAILED"/>
        <action android:name="android.app.action.ACTION_PASSWORD_SUCCEEDED"/>
      </intent-filter>
    </receiver>
```

The implementations of the corresponding onPasswordChanged(), onPasswordFailed(), and onPasswordSucceeded() methods simply display Toast messages about those events:

```
  @Override
  public void onPasswordChanged(Context ctxt, Intent intent) {
    DevicePolicyManager mgr=

(DevicePolicyManager)ctxt.getSystemService(Context.DEVICE_POLICY_SERVICE);
    int msgId;

    if (mgr.isActivePasswordSufficient()) {
      msgId=R.string.compliant;
    }
    else {
      msgId=R.string.not_compliant;
    }

    Toast.makeText(ctxt, msgId, Toast.LENGTH_LONG).show();
  }

  @Override
  public void onPasswordFailed(Context ctxt, Intent intent) {
    Toast.makeText(ctxt, R.string.password_failed, Toast.LENGTH_LONG)
         .show();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
}

@Override
public void onPasswordSucceeded(Context ctxt, Intent intent) {
  Toast.makeText(ctxt, R.string.password_success, Toast.LENGTH_LONG)
       .show();
}
```

However, these will illustrate some quirks in the behavior of the device administration APIs:

- `onPasswordSucceeded()` is not called on *every* successful password entry, only those that come after a prior `onPasswordFailed()` call. One imagines that perhaps `onPasswordSucceededAfterItHadFailedBefore()` was deemed to be too wordy.
- `isActivePasswordSufficient()` will return a value based on the *previous* password in `onPasswordChanged()`, not the newly-changed password. Since the system will prevent the user from entering a new password that is insufficient, you should not need to call `isActivePasswordSufficient()` from `onPasswordChanged()`.
- A `Toast` cannot display over the lockscreen, and so the `onPasswordFailed()` `Toast` will never be seen.

Getting Along with Others

Bear in mind that you might not be the only device administrator on any given device. If there are multiple administrators, the most secure requirements are in force. So, for example, if Admin A requests a minimum password length of 7, and Admin B requests a minimum password length of 10, the user will have to supply a password that is at least 10 characters long, to meet both device administrators' requirements.

This also means that certain requests you make may fail. For example, if you decide to say that you do *not* need encryption (`setStorageEncryption()` with a value of `false`), if something *else* needs encryption, the user will still need to encrypt their device.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

PowerManager and WakeLocks

There are going to be times when you want the device to keep running, even though it ordinarily would go into a sleep mode, with the CPU powered down and the screen turned off. Sometimes, that will be based upon user interactions, or the lack thereof, such as keeping the screen on while playing back a video. Sometimes, that will be to allow background scheduled work to run to completion, as was introduced in the chapter on `AlarmManager`.

This chapter looks a bit more at the details of this sort of power management, including coverage of how `AlarmManager` works.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on `AlarmManager`.

Keeping the Screen On, UI-Style

If your objective is to keep the screen (and CPU) on while your activity is in the foreground, the simplest solution is to add `android:keepScreenOn="true"` to something in the activity's layout. So long as that widget or container is visible, the screen will stay on.

If you wish to do this conditionally, `setKeepScreenOn()` allows you to toggle this setting at runtime.

Once your activity is no longer in the foreground, or the widget or container is no longer visible, the effect lapses, and screen operation returns to normal.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Role of the WakeLock

Most of the time in Android, you are developing code that will run while the user is actually using the device. Activities, for example, only really make sense when the device is fully awake and the user is tapping on the screen or keyboard.

Particularly with scheduled background tasks, though, you need to bear in mind that the device will eventually "go to sleep". In full sleep mode, the display, main CPU, and keyboard are all powered off, to maximize battery life. Only on a low-level system event, like an incoming phone call, will anything wake up the device.

Another thing that will partially wake up the phone is an `Intent` raised by the `AlarmManager`. So long as broadcast receivers are processing that `Intent`, the `AlarmManager` ensures the CPU will be running (though the screen and keyboard are still off). Once the broadcast receivers are done, the `AlarmManager` lets the device go back to sleep.

You can achieve the same effect in your code via a `WakeLock`.

One of the changes that the core Android team made to the Linux kernel was to introduce the concept of the "wakelock". In simple terms, a wakelock allows a Linux userland application — such as our Android SDK apps — to control whether or not the CPU can be powered down as part of a sleep mode. While a wakelock is in force, the CPU will remain on and processing instructions from the processes and threads that are on the device.

From the SDK, to access a wakelock, you use a `WakeLock` object, obtained from the `PowerManager` system service. When you call `acquire()` on that `WakeLock`, the CPU will remain on; when you call `release()` on that `WakeLock`, the CPU can fall back asleep, if there are no other outstanding `WakeLock`s from SDK apps or the operating system itself.

There are four types of `WakeLock` objects. All will keep the CPU on. They vary in their effects on the screen (leave it off, have it display with dim backlight, have it display with normal backlight) and any physical keys (ignore or accept). You will pass a flag into `newWakeLock()` on the `PowerManager` system service to indicate what type of `WakeLock` you want. The most common is the `PARTIAL_WAKE_LOCK`, which keeps the CPU on but leaves the screen and keyboard off — ideal for periodic background work triggered by an `AlarmManager` event.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What WakefulIntentService Does

For a _WAKEUP alarm, the AlarmManager will arrange for the device to stay awake, via a WakeLock, for as long as the BroadcastReceiver's onReceive() method is executing. For some situations, that may be all that is needed. However, onReceive() is called on the main application thread, and Android will kill off the receiver if it takes too long.

Your natural inclination in this case is to have the BroadcastReceiver arrange for a Service to do the long-running work on a background thread, since BroadcastReceiver objects should not be starting their own threads. Perhaps you would use an IntentService, which packages up this "start a Service to do some work in the background" pattern. And, given the preceding section, you might try acquiring a partial WakeLock at the beginning of the work and release it at the end of the work, so the CPU will keep running while your IntentService does its thing.

This strategy will work... some of the time.

The problem is that there is a gap in WakeLock coverage, as depicted in the following diagram:

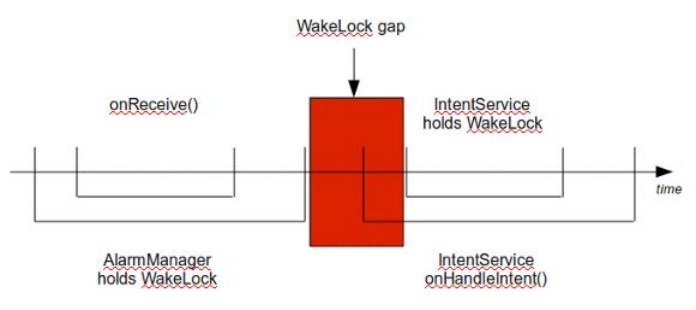


Figure 679: The WakeLock Gap

The BroadcastReceiver will call startService() to send work to the IntentService, but that service will not start up until after onReceive() ends. As a result, there is a window of time between the end of onReceive() and when your IntentService can acquire its own WakeLock. During that window, the device might fall back asleep. Sometimes it will, sometimes it will not.

What you need to do, instead, is arrange for overlapping WakeLock instances. You need to acquire a WakeLock in your BroadcastReceiver, during the onReceive()

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

execution, and hold onto that `WakeLock` until the work is completed by the `IntentService`:

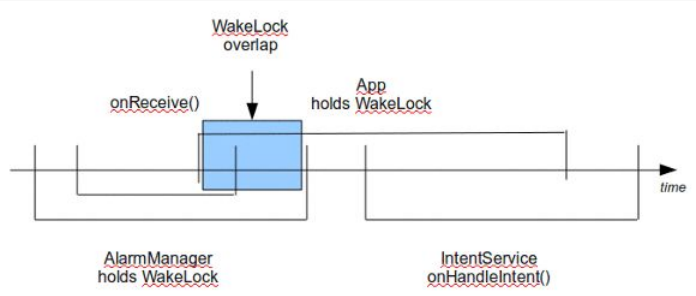


Figure 680: The WakeLock Overlap

Then you are assured that the device will stay awake as long as the work remains to be done.

The `WakefulIntentService` recipe described in its chapter does not have you manage your own `WakeLock`. That is because `WakefulIntentService` handles it for you. One reason why `WakefulIntentService` exists is to manage that `WakeLock`, because `WakeLocks` suffer from one major problem: they are not `Parcelable`, and therefore cannot be passed in an `Intent` extra. Hence, for our `BroadcastReceiver` and our `WakefulIntentService` to use the same `WakeLock`, they have to be shared via a static data member... which is icky. `WakefulIntentService` is designed to hide this icky part from you, so you do not have to worry about it.

`WakefulIntentService` also handles various edge and corner cases, such as:

- What happens if Android elects to get rid of your process due to low memory conditions?
- What happens if your `doWakefulWork()` crashes, so we do not leak the acquired `WakeLock`?
- What if your UI also sends commands to the `WakefulIntentService`, or your processing takes longer than your polling period in `AlarmManager`, so that we have more than one piece of work outstanding at a point in time?

The one requirement related to a `WakeLock` that `WakefulIntentService` imposes upon you is the `WAKE_LOCK` permission. Any code in your process that is directly manipulating `WakeLock` objects needs this permission, even if that code is from a third-party JAR like `WakefulIntentService`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

JobScheduler

`AlarmManager` was our original solution for doing work on a periodic basis. However, `AlarmManager` can readily be misused, in ways that impact the battery — this is why API Level 19 put renewed emphasis on "inexact" alarm schedules. Worse, `AlarmManager` will give us control at points in time that may be useless to us, such as giving us control when there is no Internet access, when the point of the scheduled work is to transfer some data over the Internet.

Android 5.0 introduced `JobScheduler`, which offers a more sophisticated API for handling these sorts of scenarios. This chapter will explore how to set up `JobScheduler` and use it for one-off and periodic work.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on `AlarmManager`. Also, you should have read the chapter on `PowerManager` and wakelocks.

The Limitations of AlarmManager

`AlarmManager` does its job, and frequently does it well. However, it is far from perfect:

- It does not persist its alarm schedule across reboots, forcing us to implement an `ACTION_BOOT_COMPLETED` `BroadcastReceiver` to re-establish our alarms
- It does not keep the device awake after waking it up with a `_WAKEUP` alarm, forcing us to use tools like `WakefulBroadcastReceiver` to make sure that we can get our work done without the device falling back asleep

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- It gives us control even if the work we want to do is not possible, such as wanting to download material from the Internet but being woken up at points in time when we lack a working Internet connection (e.g., a WiFi-only tablet in a location for which it does not recognize any access points)
- In cases where the criteria we want cannot be met, we cannot readily implement any sort of back-off policy, except by doing the calculations ourselves and perhaps abandoning the convenient "repeating" API outright

And so on. `AlarmManager` is nice, but it would be better to have another solution.

Enter the JobScheduler

`JobScheduler` was designed to handle those four problems outlined above:

- It persists its roster of jobs and will re-establish them automatically after a reboot. Note, though, that you still have to hold the `ACTION_BOOT_COMPLETED` permission for this to work. Also note that you do not *have* to have jobs be persisted — this is an opt-in capability of `JobScheduler`.
- It handles "wakefulness" for us, via its own `WakeLock`, so we do not have to worry about it ourselves.
- It offers an API where we can specify criteria to be satisfied before we should be given control, notably a criteria indicating that we need a working network connection.
- If our criteria cannot be met, `JobScheduler` implements a configurable back-off policy, so we can slow down our attempts to get control when those attempts are regularly failing.

Employing JobScheduler

The `JobScheduler/PowerHungry` sample project demonstrates the use of `JobScheduler`, by way of comparing its use to that of `AlarmManager`.

The UI for `JobScheduler` allows you to pick from three types of event schedules: exact alarm, inexact alarm, and `JobScheduler`. You can also choose from one of four polling periods: 1 minute, 15 minutes, 30 minutes, and 60 minutes:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

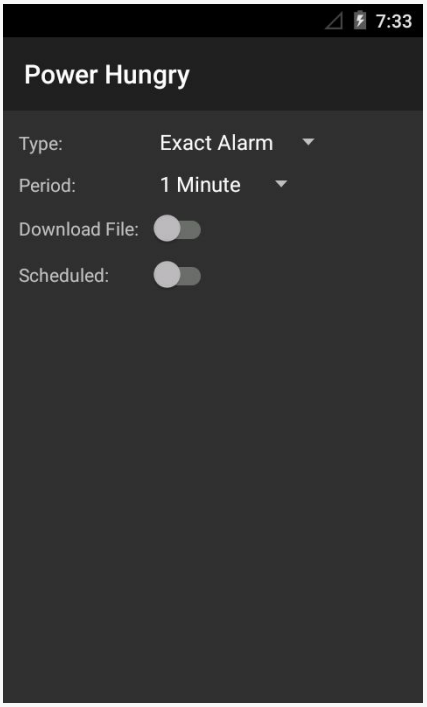


Figure 681: PowerHungry Demo, As Initially Launched

A `Switch` (in its `Theme.Material` styling) allows you to determine whether you are simply getting control at those points in time to just log to LogCat, or whether you are going to try to do some work at those points in time. Specifically, the "work" is to download a file, using `HttpUrlConnection`.

The bottom `Switch` toggles on and off the event schedules. When the event schedules are toggled on, you cannot manipulate the rest of the UI — you need to turn off the events in order to change the event configuration.

Note that none of this information is persisted. This is a lightweight demo; it is expected that you are keeping this UI in the foreground while a test is running.

Defining and Scheduling the Job

The "job" is defined as an instance of `JobInfo`, typically created using an instance of `JobInfo.Builder` to configure a `JobInfo` using a fluent builder-style API. We teach the `JobInfo` the work to do and when to do it, then use a `JobScheduler` to actually schedule the job.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In the sample app, this work is mostly accomplished via a `manageJobScheduler()` method on the `MainActivity` class:

```
private void manageJobScheduler(boolean start) {
  if (start) {
    JobInfo.Builder b=new JobInfo.Builder(JOB_ID,
        new ComponentName(this, DemoJobService.class));
    PersistableBundle pb=new PersistableBundle();

    if (download.isChecked()) {
      pb.putInt(KEY_DOWNLOAD_AS_AN_INT, 1);
      b.setExtras(pb).setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY);
    } else {
      b.setRequiredNetworkType(JobInfo.NETWORK_TYPE_NONE);
    }

    b.setPeriodic(getPeriod()).setPersisted(false)
        .setRequiresCharging(false).setRequiresDeviceIdle(true);

    jobs.schedule(b.build());
  }
  else {
    jobs.cancel(JOB_ID);
  }
}
```

The `start` parameter to `manageJobScheduler()` is driven by the bottom `Switch` widget. A `start` value of `true` means that we should start up the job; a value of `false` means that we should cancel any existing job.

If `start` is `true`, we begin by creating a `JobInfo.Builder`, supplying two key pieces of data:

- an `int` that will serve as the job ID, which needs to be unique to our app but does not have to be unique for the whole device
- a `ComponentName` identifying the `JobService` that will actually implement the work of the job itself

The primary way of passing data from the scheduling code (our activity) and the job-implementing code (`JobService`) is by means of a `PersistableBundle` – a `Bundle`-like object that can be persisted to disk. What we would like to do is pass a `boolean` in this `PersistableBundle` to indicate whether we should be downloading a file or not, based on the first `Switch` widget. Alas, for inexplicable reasons, `PersistableBundle` does not offer accessors for a `boolean`. So, we settle for an `int`, where a non-zero value indicates that we want to download a file. Our `PersistableBundle` can have more data than just this one extra, though that is all

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

we need in this case. We attach the PersistableBundle to the JobInfo via the setExtras() method on the JobInfo.Builder.

We can also call methods on the JobInfo.Builder to configure the criteria that should be satisfied before giving us control. In our case, one criterion that we need is to have a network connection, but only if we are supposed to be downloading a file. So, we call setRequiredNetworkType() in either case, indicating that we either want ANY type of network connection (metered or unmetered) or NONE.

Other criteria-defining methods that we invoke include setRequiresCharging() (set to false to indicate we want control even if we are on battery) and setRequiresDeviceIdle() (set to true to indicate that we want control only if the user is not using it).

In the case of this sample, we want to do this work every so often, based upon the period chosen by the user in the bottom Spinner and retrieved via the getPeriod() method. So, we call setPeriodic() on the JobInfo.Builder to request getting control with that frequency, bearing in mind that this is merely a hint, not a requirement, and we may get control more or less frequently than this.

We also call setPersisted(false) to indicate that we do not need for this job to be persisted, so it will be lost on a reboot. If we instead called setPersisted(true), the manifest would need to request the RECEIVE_BOOT_COMPLETED permission to have the job be re-created at boot time.

Finally, we call schedule() on a JobScheduler instance named jobs to schedule the job.

The jobs data member is populated up in onCreate() of the activity:

```
    jobs=(JobScheduler)getSystemService(JOB_SCHEDULER_SERVICE);
```

While this may look like an ordinary request for a system service, it has one issue: your version of Android Studio may not know anything about it. In that case, this code will fail a Lint check, complaining that the service is not recognized. This is a bug that should be fixed in Android Studio 0.8.14. The workaround is to add @SuppressWarnings("ResourceType") to the method where you are making this getSystemService() call to suppress this Lint check.

If the start parameter to manageJobScheduler() is false, we call cancel() on the JobScheduler, passing in our unique job ID (JOB_ID) to indicate what job to cancel.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Or, we could have called `cancelAll()`, which would cancel all jobs scheduled by our application.

Implementing the Job

The work for the job itself is handled by a `JobService`. This is a subclass of `Service` that we, in turn, extend ourselves, overriding two job-specific callback methods to actually do the work: `onStartJob()` and `onStopJob()`.

The `JobService` in our sample app is `DemoJobService`:

```
package com.commonsware.android.job;

import android.app.job.JobParameters;
import android.app.job.JobService;
import android.os.PersistableBundle;
import android.util.Log;

public class DemoJobService extends JobService {
  private volatile Thread job=null;

  @Override
  public boolean onStartJob(JobParameters params) {
    PersistableBundle pb=params.getExtras();

    if (pb.getInt(MainActivity.KEY_DOWNLOAD_AS_AN_INT, 0)==1) {
      job=new DownloadThread(params);
      job.start();

      return(true);
    }

    Log.d(getClass().getSimpleName(), "job invoked");

    return(false);
  }

  @Override
  synchronized public boolean onStopJob(JobParameters params) {
    if (job!=null) {
      Log.d(getClass().getSimpleName(), "job interrupted");
      job.interrupt();
    }

    return(false);
  }

  synchronized private void clearJob() {
    job=null;
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private class DownloadThread extends Thread {
    private final JobParameters params;

    DownloadThread(JobParameters params) {
      this.params=params;
    }

    @Override
    public void run() {
      Log.d(getClass().getSimpleName(), "job begins");
      new DownloadJob().run();
      Log.d(getClass().getSimpleName(), "job ends");
      clearJob();
      jobFinished(params, false);
    }
  }
}
```

`onStartJob()` is passed a `JobParameters`. This serves both as a "handle" identifying a particular job invocation and giving us access to the job ID (`getJobId()`) and `PersistableBundle` of extras (`getExtras()`) that were set up by our `JobInfo` when we scheduled the job.

`onStartJob()` needs to return `true` if we have successfully forked a background thread to do the work, or `false` if no work needs to be done. In our case, this is determined by whether or not we want to try to download a file. In a production-grade app, this may be determined by whether there is any work to be done (e.g., "do we have entries in the upload queue?").

In `onStartJob()`, we check the `PersistableBundle` to see if we are supposed to download a file. If we are, we fork a `DownloadThread` to do that work, then return `true`. Otherwise, we return `false`.

Because this sample app illustrates the difference in behavior between `JobScheduler` and `AlarmService`, we want to isolate the actual download-the-file logic into a common implementation that can be used from either code path. That takes the form of a `DownloadJob`, which implements `Runnable` and does the download work when it is `run()`:

```
package com.commonsware.android.job;

import android.net.Uri;
import android.os.Environment;
import android.util.Log;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

class DownloadJob implements Runnable {
  static final Uri TO_DOWNLOAD=
      Uri.parse("https://commonsware.com/Android/excerpt.pdf");

  @Override
  public void run() {
    try {
      File root=

Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);

      root.mkdirs();

      File output=new File(root, TO_DOWNLOAD.getLastPathSegment());

      if (output.exists()) {
        output.delete();
      }

      URL url=new URL(TO_DOWNLOAD.toString());
      HttpURLConnection c=(HttpURLConnection)url.openConnection();

      FileOutputStream fos=new FileOutputStream(output.getPath());
      BufferedOutputStream out=new BufferedOutputStream(fos);

      try {
        InputStream in=c.getInputStream();
        byte[] buffer=new byte[8192];
        int len=0;

        while ((len=in.read(buffer)) >= 0) {
          out.write(buffer, 0, len);
        }

        out.flush();
      }
      finally {
        fos.getFD().sync();
        out.close();
        c.disconnect();
      }
    }
    catch (IOException e2) {
      Log.e("DownloadJob", "Exception in download", e2);
    }
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

DownloadThread delegates to DownloadJob to do the actual work. However, when the work is complete, it then calls jobFinished() on the DemoJobService. jobFinished(), as the name suggests, tells the framework that we are finished doing the work associated with this job. If the job succeeded, we pass false as the second parameter, to indicate that this job does not need to be rescheduled. If, on the other hand, we were unable to actually do the work (e.g., we cannot connect to the desired server, perhaps due to server maintenance), we would pass true as the second parameter, to request that this job be rescheduled to be invoked again shortly, so that we can retry the operation.

Our onStopJob() method will be called by Android if environmental conditions have changed and we should stop the background work that we are doing. For example, we asked to do this work when the device was idle — if the user picks up the device and starts using it, we should stop our background work. In this case, if the job thread is still outstanding, we cancel() it. onStopJob() should return true if this job is still needed and should be retried, or false otherwise. Most short-period periodic jobs should return false, to just worry about the next job in the next period, and that is what onStopJob() does here. One-time jobs, or jobs with long periods (e.g., a day), may wish to return true to ensure that they will get another chance to do the desired work. We will cover more about this issue later in this chapter.

Wiring in the Job Service

Since a JobService is a Service, we need the corresponding <service> element in the manifest. For a JobService, the <service> element is perfectly normal... with one exception:

```
        <service
            android:name=".DemoJobService"
            android:permission="android.permission.BIND_JOB_SERVICE"
        />
```

You need to defend the service with the BIND_JOB_SERVICE permission. This only allows code that holds the BIND_JOB_SERVICE permission to start or bind to this service, which should limit it to the OS itself.

The Rest of the Sample

As noted earlier, the UI for our activity is a pair of Spinner widgets, along with a pair of Switch widgets:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<?xml version="1.0" encoding="utf-8"?>

<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="8dp"
    android:useDefaultMargins="true">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/type_label"
        android:layout_row="0"
        android:layout_column="0"/>

    <Spinner
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/type"
        android:layout_row="0"
        android:layout_column="1"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/period_label"
        android:layout_row="1"
        android:layout_column="0"/>

    <Spinner
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/period"
        android:layout_row="1"
        android:layout_column="1"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/download_label"
        android:layout_row="2"
        android:layout_column="0"/>

    <Switch
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/download"
        android:layout_row="2"
        android:layout_column="1"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        android:text="@string/scheduled_label"
        android:layout_row="3"
        android:layout_column="0"/>

    <Switch
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/scheduled"
        android:layout_row="3"
        android:layout_column="1"/>
</GridLayout>
```

onCreate() of MainActivity sets up the UI, including populating the two Spinner widgets based on <string-array> resources and hooking up the activity to respond to changes in the checked state of the scheduled Switch widget:

```
  @SuppressWarnings("ResourceType")
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);
    type=(Spinner)findViewById(R.id.type);

    ArrayAdapter<String> types=
        new ArrayAdapter<String>(this,
            android.R.layout.simple_spinner_item,
            getResources().getStringArray(R.array.types));

    types.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
    type.setAdapter(types);

    period=(Spinner)findViewById(R.id.period);

    ArrayAdapter<String> periods=
        new ArrayAdapter<String>(this,
            android.R.layout.simple_spinner_item,
            getResources().getStringArray(R.array.periods));


periods.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
    period.setAdapter(periods);

    download=(Switch)findViewById(R.id.download);
    scheduled=(Switch)findViewById(R.id.scheduled);
    scheduled.setOnCheckedChangeListener(this);

    alarms=(AlarmManager)getSystemService(ALARM_SERVICE);
    jobs=(JobScheduler)getSystemService(JOB_SCHEDULER_SERVICE);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

When the user toggles the `scheduled` `Switch` widget, we examine the `type` `Spinner` and route control to a method dedicated for handling that particular type of periodic request, such as the `manageJobScheduler()` method we saw earlier in this chapter:

```
  @Override
  public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
    toggleWidgets(!isChecked);

    switch(type.getSelectedItemPosition()) {
      case 0:
        manageExact(isChecked);
        break;

      case 1:
        manageInexact(isChecked);
        break;

      case 2:
        manageJobScheduler(isChecked);
        break;
    }
  }
```

Our `onCheckedChanged()` for the `schedule` `Switch` also calls a `toggleWidgets()` method that enables or disables the other widgets, depending upon whether the `schedule` `Switch` is checked or unchecked:

```
  private void toggleWidgets(boolean enable) {
    type.setEnabled(enable);
    period.setEnabled(enable);
    download.setEnabled(enable);
  }
```

If the user had chosen an exact alarm, `onCheckedChanged()` routes control to `manageExact()`:

```
  private void manageExact(boolean start) {
    if (start) {
      long period=getPeriod();

      PollReceiver.scheduleExactAlarm(this, alarms, period,
          download.isChecked());
    }
    else {
      PollReceiver.cancelAlarm(this, alarms);
    }
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It, in turn, routes control over to a `PollReceiver`, a `WakefulBroadcastReceiver` that is set up for handling our alarms:

```
package com.commonsware.android.job;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;
import android.support.v4.content.WakefulBroadcastReceiver;

public class PollReceiver extends WakefulBroadcastReceiver {
  static final String EXTRA_PERIOD="period";
  static final String EXTRA_IS_DOWNLOAD="isDownload";

  @Override
  public void onReceive(Context ctxt, Intent i) {
    boolean isDownload=i.getBooleanExtra(EXTRA_IS_DOWNLOAD, false);
    startWakefulService(ctxt,
        new Intent(ctxt, DemoScheduledService.class)
            .putExtra(EXTRA_IS_DOWNLOAD, isDownload));

    long period=i.getLongExtra(EXTRA_PERIOD, -1);

    if (period>0) {
      scheduleExactAlarm(ctxt,
          (AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE),
          period, isDownload);
    }
  }

  static void scheduleExactAlarm(Context ctxt, AlarmManager alarms,
                                 long period, boolean isDownload) {
    Intent i=new Intent(ctxt, PollReceiver.class)
        .putExtra(EXTRA_PERIOD, period)
        .putExtra(EXTRA_IS_DOWNLOAD, isDownload);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);

    alarms.setExact(AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime()+period, pi);
  }

  static void scheduleInexactAlarm(Context ctxt, AlarmManager alarms,
                                   long period, boolean isDownload) {
    Intent i=new Intent(ctxt, PollReceiver.class)
        .putExtra(EXTRA_IS_DOWNLOAD, isDownload);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);

    alarms.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime()+period, period, pi);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  static void cancelAlarm(Context ctxt, AlarmManager alarms) {
    Intent i=new Intent(ctxt, PollReceiver.class);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);

    alarms.cancel(pi);
  }
}
```

This sample app has a `targetSdkVersion` of 21. Hence, on Android 5.0 devices — the ones that have `JobScheduler`, we cannot set up exact repeating alarms. Our only option is to handle the repeating work ourselves.

Hence, `scheduleExactAlarm()` creates a broadcast `PendingIntent`, on an `Intent` pointing at our `PollReceiver`, with a pair of extras indicating the polling period and whether or not we should be downloading a file. It then uses `setExact()` on an `AlarmManager` to schedule a *one-off* event to occur one polling period from now.

That, in turn, will trigger `onReceive()` of the `PollReceiver`. Here, we call `startWakefulService()` to have our work be done by a `DemoScheduledService`. In addition, if we have a polling period, that means that this is an exact alarm, and we call `scheduleExactAlarm()` to set up the next occurrence of this "repeating" event.

`DemoScheduledService` is simply an `IntentService` wrapper around the `DownloadJob` that we used with `DemoJobService`, logging the fact that it ran and calling `completeWakefulIntent()` to indicate that the work initiated by the `WakefulBroadcastReceiver` was done.

`cancelAlarm()` on `PollReceiver` — called by `manageExact()` when we are stopping the repeating event — creates an equivalent `PendingIntent` to the ones used for the `AlarmManager` events, and uses that with `cancel()` on `AlarmManager` to cancel those events.

If the user had chosen an exact alarm, `onCheckedChanged()` routes control to `manageInexact()`:

```
  private void manageInexact(boolean start) {
    if (start) {
      long period=getPeriod();

      PollReceiver.scheduleInexactAlarm(this, alarms, period,
          download.isChecked());
    }
    else {
      PollReceiver.cancelAlarm(this, alarms);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }
```

It uses the same recipe as `manageExact()`, except that it calls `scheduleInexactAlarm()` on `PollReceiver`. `scheduleInexactAlarm()`, in turn, uses `setInexactRepeating()` on `AlarmManager` to arrange to get control every so often.

Pondering Backoff Criteria

Sometimes, even with the Internet-availability checks offered by `JobScheduler`, you find that you cannot actually do the job you scheduled. Perhaps the server is down for maintenance, or has been replaced by a honeycomb frame, or something. In this case, while you failed to do the job now, you may want to try again later.

Sometimes, "later" can just be handled by your existing `JobScheduler` setup. If the job in question is a periodic job, and missing a whole period is not a big problem, you might just continue on normally.

However, sometimes you will want the job to be retried, either because:

- it was a one-shot job, not a periodic one, or
- the period of the job is fairly long (e.g., once per day) and you want to retry well before the job is scheduled to happen again

Requesting that a job be retried is handled by the `boolean` parameter to `jobFinished()` or the `boolean` return value from `onStopJob()`. `true` means that you want the job to be rescheduled; `false` means that it is OK to skip the job entirely.

Given that you use `true` for either `jobFinished()` or `onStopJob()`, there are three possible options for how to request and retry a failed job:

- What happens for a job which you requested only run when the device is idle
- What happens for other jobs by default
- How you can influence the timing of when the job is retried, known as the "backoff criteria"

Idle Jobs

If you requested idle-only jobs, if the user wakes up the device while the job is going on, you will be called with `onStopJob()`. Ideally, you then stop the background work

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

and return `true` or `false` from `onStopJob()` to determine if the job should be rescheduled.

If you request a job be rescheduled, when that job is set up to only run when the device is idle, the job is simply "put back in the queue" to be tried again during the next idle window.

Default Behavior

If, for a non-idle-only job, you use `true` for `jobFinished()` or `onStopJob()`, the next time to try will be calculated using the default backoff criteria, which has a time of 30 seconds and a policy of `BACKOFF_POLICY_EXPONENTIAL`.

What this means is that the first time you use `true`, your job will be tried again 30 seconds later. If you use `true` again for that job, it will be tried again 60 seconds later. If you use `true` again, it would be tried 120 seconds later — in other words, each job failure will reschedule using the formula `2^(n-1)*t`, where `n` is the number of failures and `t` is 30 seconds.

However, there is a cap of 18,000,000 milliseconds, or what normal people would refer to as "5 hours". That is the most your job will be delayed, regardless of how many failures you have.

Custom Backoff Criteria

You can change the backoff criteria for non-idle-only jobs via a call to `setBackoffCriteria()` on your `JobInfo.Builder`, where you provide your own time (measured in milliseconds) and policy (`BACKOFF_POLICY_EXPONENTIAL` or `BACKOFF_POLICY_LINEAR`).

As noted above, the formula for exponential backoff rescheduling is **2[sup>n–1</sup]t**, where `n` is the number of failures and `t` is your chosen time.

The formula for linear backoff rescheduling is `n*t`, where `n` is the number of failures and `t` is your chosen time.

Other JobScheduler Features

There are a few other options for scheduling jobs that may be of use to you in select circumstances:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `JobInfo.Builder` has `setOverrideDeadline()`, which indicates a maximum delay for this job before it will be executed even if other criteria (e.g., idleness) have been met. Note that this is only available on one-shot jobs, not periodic jobs.
- The `JobParameters` passed to `onStartJob()` has an `isOverrideDeadlineExpired()` method. This will return `true` if the job was executed early due to a `setOverrideDeadline()` value being met. This will indicate to you that your requirements may not be met (e.g., Internet access) and you will need to double-check those things yourself.
- `JobInfo.Builder` has `setMinimumLatency()` which sets a minimum delay time; the job will not be considered until at least this amount of time has elapsed. Note that this is only available on one-shot jobs, not periodic jobs.

Also, `JobScheduler` has a `getAllPendingJobs()` method, that returns a `List` of `JobInfo` objects representing "the jobs registered by this package that have not yet been executed". Presumably, this includes the next occurrence of any periodic jobs and any jobs that are blocked pending a backoff delay, though the documentation is unclear on this point.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Push Notifications with GCM

Google Cloud Messaging – GCM for short — is Google's current framework for asynchronously delivering notifications from the Internet ("cloud") to Android devices. Rather than the device waking up and polling on a regular basis at the behest of your app, your app can register for notifications and then wait for them to arrive. GCM is engineered with power savings in mind, aiming to minimize the length of time 3G radios are exchanging data.

The proper use of GCM means better battery life for your users. It can also reduce the amount of time your code runs, which helps you stay out of sight of users looking to pounce on background tasks and eradicate them with task killers.

GCM replaces C2DM ("cloud to device messaging") as Google's push framework for Android. While C2DM is still in operation, it is accepting no new developer registrations. New development should use GCM; apps already using C2DM should plan to cut over to GCM as soon as is practical.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapters on:

- broadcast Intents
- service theory
- `AlarmManager` (though mostly for the discussion of `WakefulIntentService`)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Precursor: C2DM

C2DM debuted in 2010 as a way for apps to receive "push" messages: messages sent to it from "the cloud" by the app developer's server. It quickly became popular, for everything from triggering near-real-time data synchronization (e.g., Remember the Milk to-do list updates) to lightweight coordination between multiple players in a game.

However, C2DM was a Google Labs product and in perpetual beta form. When Google Labs was shut down, C2DM was in limbo: not canceled, but not converted into an actual product. Most likely, that was because while they knew the concept was sound, they wanted to tweak the implementation and APIs and were not ready with its replacement just yet.

The Replacement: GCM

GCM replaced C2DM in 2012, with C2DM moving into a deprecated state, continuing to function but accepting no new applications.

GCM follows the same basic structure as C2DM, with the app registering for messages, and the developer's server sending messages to the app via a Google-supplied Web service interface. Hence, apps written to use C2DM should migrate over to GCM without significant architectural changes.

GCM raises the 1KB limit for the message to 4KB. It also simplifies the server side, by replacing a fairly clunky authentication model with a "Simple API Key" and lifting quotas that had hamstrung popular C2DM-enabled apps. Also, if the same message needs to be delivered to multiple devices, GCM can send to up to 1,000 at a time, whereas C2DM was limited to a one-device-per-Web-service-call model.

As a result, GCM provides the benefits of C2DM without some of the annoying limitations. That being said, GCM is not perfect, and we will examine some of the limitations that remain later in this chapter.

The Re-Replacement: GCM 2013

At Google I|O 2013, Google debuted a substantial update to GCM. The update enables a handful of new features, most notably an upstream messaging option, so devices can "push" messages to servers just as servers can push messages to clients.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Along the way, Google:

- Added a dependency to the Play Services SDK library project
- Deprecated a JAR containing code that you need, in one form or fashion, to write GCM-enabled apps, forcing you to basically rewrite lots of that JAR from scratch or "forward-port" the old implementation
- Further weakened their documentation, including all but eliminating any discussion of actual server implementations

The Pieces of Push

There are many components that you will need to work with in order to enable GCM in your app, both inside the app itself and in your app server (or other off-app environment) where you are trying to push messages to the app.

A Suitable Android Environment

GCM is part of Play Services. That means that you probably want to test your app on hardware that legitimately has the Play Store on it. Some emulator images, offering "Google APIs", may have a copy of Play Services on it, though this can sometimes be out of date or otherwise buggy.

API Key

Since GCM is a Google service, you need a Google API key to use it. To get one of these:

1. Log into https://cloud.google.com/console with your Google account
2. In the navigation on the left, go into "APIs & auth > APIs"
3. Find "Google Cloud Messaging for Android" and click the "OFF" button next to it (note: this might involve then agreeing to additional terms of service)
4. In the navigation on the left, go into "APIs & auth > Registered apps"
5. Click the red REGISTER APP button.
6. Give the app a name and choose Web Application, then click Register
7. Click the "Server Key" section for your server key

To confirm that your API key works, if you have the `curl` program, the `gcmtest` script in the `Push/GCMClient` sample project will help you confirm that the API key works. Run `gcmtest`, with your API key as a parameter. If you get something like the following, your API key is good:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
{ "multicast_id":7932441338082226994,
  "success":0,
  "failure":1,
  "canonical_ids":0,
  "results":[{"error":"InvalidRegistration"}]}
```

If, instead, you get an HTTP 401 error, then your API key is flawed in some way.

Play Services Framework

In the initial incarnation of GCM, the scaffolding for your client-side code came in the form of a JAR, accompanied by source code, shipped as part of the "Google Cloud Messaging for Android Library" entry in the SDK Manager.

The 2013 update to GCM officially deprecated that JAR, in favor of a new `GoogleCloudMessaging` object from the Play Services SDK.

Android Studio users should install the "Google Repository" entry in the SDK Manager. At that point, you can add a dependency upon the `com.google.android.gms:play-services-base` artifact for some appropriate version, such as `com.google.android.gms:play-services-base:6.5.87`.

Eclipse users will need to install the "Google Play services" entry in the SDK Manager, then add to your project a reference to the Android library project found in the `extras/google/google_play_services/libproject/google-play-services_lib/` directory in your Android SDK installation.

You will find reference to an older "Google Cloud Messaging for Android Library" in the SDK Manager. Nowadays, it will only appear there if you have the "Obsolete" checkbox checked. This is the original, pre-Play Services library for accessing GCM. The sample GCM client app profiled in this book demonstrates how some of that old code can be blended in with some of the new Google-supplied GCM samples to create a framework that minimizes the amount of GCM-specific code needed in your app. However, since the sample project contains all of the forked code, you do not need this library, unless you are maintaining some legacy GCM code that still uses it.

Android App

Beyond the aforementioned library, your application using GCM has several requirements that you must meet, outlined in the following sections.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Custom Permission

Your application needs to define a custom permission, whose name is your application's package name with `.permission.C2D_MESSAGE` appended to the end. So, for an app residing in the `com.commonsware.android.gcm.client` package, the permission to be defined is:

```
<permission
  android:name="com.commonsware.android.gcm.client.permission.C2D_MESSAGE"
  android:protectionLevel="signature"/>
```

You also need the corresponding `<uses-permission>` element:

```
<uses-permission
  android:name="com.commonsware.android.gcm.client.permission.C2D_MESSAGE"/>
```

This is all rather odd, considering that nothing else in your project seems to refer to this permission. Also, it is not required if you are only supporting API Level 16 and higher (i.e., your `android:minSdkVersion` is set to at least 16).

Additional Permissions

There are several other permissions that you need to hold:

```
<uses-permission android:name="com.google.android.c2dm.permission.RECEIVE"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.GET_ACCOUNTS"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
```

The first allows you to receive messages from GCM, though it uses `c2dm` in the name for backwards compatibility with C2DM. The second allows your GCM library to access the Internet, while the third allows your GCM library to access information about your Google account. The last permission allows you to hold a `WakeLock` to keep the device awake for a bit — we will use this with a `WakefulIntentService` to process incoming messages.

Your Registration Code

Your app will need to use the static `register()` method on the `GoogleCloudMessaging` class supplied by the Play Services SDK to register with GCM and create a registration ID. That ID, in turn, will be used by your server code to uniquely identify a copy of your app running on a specific device. Messages sent by your server to that registration ID will be delivered to your app on that device.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`register()` is a blocking call, doing network I/O, so you will want to register on a background thread, such as an `AsyncTask`.

You will need to send that registration ID to your server — or whatever else will be pushing messages — by some means. A typical implementation would have your app call some Web service of yours to hand over the registration ID, perhaps tying it to some user account on your Web server. This sort of network I/O also should be done in a background thread.

Your registration ID is good until you update your app to a new version, as determined via its `android:versionCode` in the manifest. Since `register()` will generate a new registration ID each time, you will want to cache the registration ID until such time as you ship a new version of the app, at which time you will need to generate a fresh ID. Google's sample code in their documentation demonstrates this.

Google's sample code also deals with the scenario in which your server somehow loses the registration ID, generating a fresh registration ID every week. Whether this is truly necessary, or whether you want to do something else to deal with this use case, is up to you.

You can place this code wherever makes sense, such as your launcher activity.

Of course, pretty much all of this used to be handled for us by Google's GCM client library. They deprecated that library and failed to replace it in the 2013 updates to GCM. The sample app, presented later in this chapter, will show some replacement code for that library, synthesized from the old library source code and the newer GCM samples.

GCM BroadcastReceiver

You will need to implement a `BroadcastReceiver` that will get control when messages are received that have been pushed to the device. Specifically:

- The receiver should be registered in the manifest, as push messages can be delivered at any point in time
- The receiver must have an `<intent-filter>` with an `<action>` of `com.google.android.c2dm.intent.RECEIVE` and a `<category>` whose value is the package name of your app
- The receiver should require the `com.google.android.c2dm.permission.SEND` permission, to prevent other apps from spoofing push messages

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You will wind up with a <receiver> element reminiscent of the following:

```
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
```

The Intent that comes into your receiver's onReceive() method will represent a push message from your server, or possibly a system message. You can call getMessageType() on a GoogleCloudMessaging instance to help decipher what sort of event occurred:

- GoogleCloudMessaging.MESSAGE_TYPE_SEND_ERROR means that some error occurred
- GoogleCloudMessaging.MESSAGE_TYPE_DELETED means that too many messages had been queued up for your app on this device (e.g., due to a protracted period of being in airplane mode), and some were deleted
- GoogleCloudMessaging.MESSAGE_TYPE_MESSAGE means that the Intent is an actual push message from the server

The extras Bundle that is part of the Intent (retrieved via getExtras()) will have name-value pairs supplied by your server, representing whatever data the server wanted to push to your app on this device.

It is up to you to do to something useful with these events. However, onReceive() of a manifest-registered BroadcastReceiver is a precarious place:

- You are running on the main application thread, so you cannot take much time
- You cannot do much of anything that lives beyond the end of onReceive(), such as forking a thread, as Android can terminate your process at will once onReceive() returns

Hence, a typical pattern is for the receiver to delegate the message to an IntentService for actual processing. And, once again, this used to be done for you by the GCM client library, which supplied a stock BroadcastReceiver and a base IntentService class that you could use. The sample app profiled in this chapter includes replacements for those now-lost classes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Your Server (a.k.a., the Thing Doing the Pushing)

Something, outside of your app, is going to be pushing messages to devices that are running your app and registered for such messages. In the GCM documentation, this is referred to as your "server".

Technically speaking, this does not have to be a "server": something running constantly with some sort of inbound socket connection. To actually push messages, you will need Internet access to Google's servers, but whether you push the message from a true server, or a desktop app, or a command-line program, is really up to you.

The classic mode for sending push messages to the device involves your server using HTTP and a REST protocol to hand the messages to Google's servers, which in turn forward the messages to devices. There is also a new Cloud Connection Service option that uses XMPP instead of HTTP, to avoid creating and tearing down sockets quite so frequently. Cloud Connection Service will be discussed later in this chapter.

GCM supplies a "server" JAR file that handles the REST protocol and gives you Java objects for building and sending the message. This JAR file has an undocumented dependency on the `json-simple` library. If your desired "server" environment is not in Java, you could re-implement the REST protocol library in some other programming language (e.g., Ruby) if desired. The GCM documentation describes the request and response formats that your library would need to handle.

For simplicity, the sample app for this chapter will demonstrate a crude command-line Java client, based upon the GCM "server" JAR, rather than expecting you to run some Java-capable Web server just to experiment with GCM.

Google's Server and the Google Services Framework

One piece of the environment that you do not control is Google's GCM server farm. When you send a message to the clients, you do not do so directly, but rather you send your message to Google by means of a REST-style Web service (or XMPP using the Cloud Connection Service). Google, in turn, forwards your messages along to the clients. Google supplies a JAR file that you can use on your server, or you could implement the REST-style interface in other ways, using anything from `curl` to a Ruby gem, if you so choose.

Each device maintains a long-running persistent socket connection to Google's GCM server farm. When your server sends a message, it passes the message to Google's GCM server farm, which finds the connection to the client(s) and sends along the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

message. If a client is unavailable for some reason, the GCM servers will cache your message for a period of time in hopes of being able to deliver it shortly.

The fact that Google servers have your message for any length of time introduces some privacy and security issues, which we will examine later in this chapter.

A Simple Push

With all this in mind, we can walk through an example of implementing GCM. This sample comes in two parts:

1. An app that runs on the Android device that uses GCM
2. A command-line Java app that can send messages via GCM, useful for light testing

The Client

Our client-side Android app can be found in the `Push/GCMClient2` sample project.

Our manifest has all the things outlined earlier in this chapter, including the custom permission, all those other permission elements, and a suitably-configured `BroadcastReceiver`:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.gcm.client"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="13"
    android:targetSdkVersion="17"/>

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"
    android:xlargeScreens="true"/>

<!--
  <permission
    android:name="com.commonsware.android.gcm.client.permission.C2D_MESSAGE"
    android:protectionLevel="signature"/>

  <uses-permission
android:name="com.commonsware.android.gcm.client.permission.C2D_MESSAGE"/>
-->
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <permission
    android:name="${applicationId}.permission.C2D_MESSAGE"
    android:protectionLevel="signature" />
  <uses-permission
    android:name="${applicationId}.permission.C2D_MESSAGE" />

  <uses-permission android:name="com.google.android.c2dm.permission.RECEIVE"/>
  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.GET_ACCOUNTS"/>
  <uses-permission android:name="android.permission.WAKE_LOCK"/>

  <application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>

    <receiver
      android:name="GCMBroadcastReceiverCompat"
      android:permission="com.google.android.c2dm.permission.SEND">
      <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE"/>

        <category android:name="${applicationId}" />
      </intent-filter>
    </receiver>

    <service android:name=".GCMIntentService"/>

    <meta-data
      android:name="com.google.android.gms.version"
      android:value="@integer/google_play_services_version"/>
  </application>

</manifest>
```

The rest of the app is made up of an activity (`MainActivity`) and an `IntentService` (`GCMIntentService`) that represent the app's "business logic", plus three additional classes that represent a partial replacement for the code from the deprecated GCM client JAR.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Activity

`MainActivity` has a highly sophisticated user interface, consisting of one really large "Register me!" button. The objective of the UI is simply to register the app with GCM, so that you can test sending messages to your app on the test device.

To do this, `MainActivity` has GCM-related hooks in `onCreate()` and the `onClick()` handler being used for the `Button`:

```
package com.commonsware.android.gcm.client;

import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Toast;

public class MainActivity extends Activity {
  static final String SENDER_ID="this is so fake"; // change
                                                   // me!

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    GCMRegistrarCompat.checkDevice(this);

    if (BuildConfig.DEBUG) {
      GCMRegistrarCompat.checkManifest(this);
    }
  }

  public void onClick(View v) {
    final String regId=GCMRegistrarCompat.getRegistrationId(this);

    if (regId.length() == 0) {
      new RegisterTask(this).execute(SENDER_ID);
    }
    else {
      Log.d(getClass().getSimpleName(), "Existing registration: "
          + regId);
      Toast.makeText(this, regId, Toast.LENGTH_LONG).show();
    }
  }

  private static class RegisterTask extends
      GCMRegistrarCompat.BaseRegisterTask {

    RegisterTask(Context context) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      super(context);
    }

    @Override
    public void onPostExecute(String regid) {
      Log.d(getClass().getSimpleName(), "registered as: " + regid);
      Toast.makeText(context, regid, Toast.LENGTH_LONG).show();
    }
  }
}
```

Specifically:

- `onCreate()` calls `checkDevice()` and `checkManifest()` static methods on a `GCMRegistrarCompat` class
- `onClick()` tries to retrieve the current registration ID via a call to the static `getRegistrationId()` method on `GCMRegistrarCompat`, then kicks off a `RegisterTask` subclass of `GCMRegistrarCompat.BaseRegisterTask` if we do not have a current valid registration ID

The `SENDER_ID` value, as is noted in the code comment, needs to be replaced with *your* sender ID. This is referred to as the "project number" in the Google Developers Console:

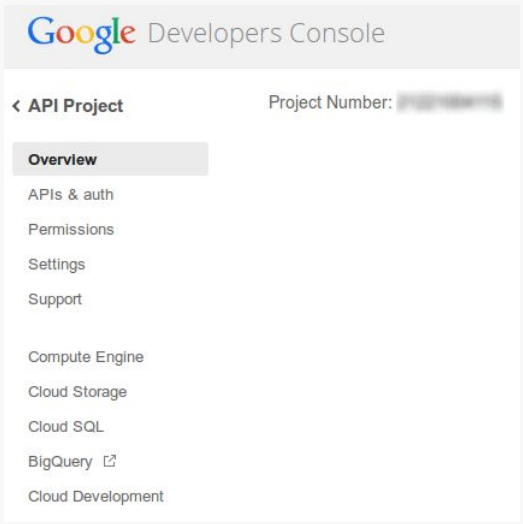


Figure 682: Google Developers Console, Showing Project Number (a.k.a., Sender ID)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

GCMRegistrarCompat

The deprecated GCM client JAR contained a class named `GCMRegistrar` that handled:

- the communication to the GCM servers for the actual registration process, and
- caching the registration ID on the device, until such time as it was known to be invalid (e.g., after an app version upgrade)

Since that JAR is deprecated, but we still need to do this work, this project contains `GCMRegistrarCompat`, which incorporates some of the original `GCMRegistrar` logic, updated to use the new `GoogleCloudMessaging` interface. And, as one might expect, the resulting code is a bit nasty.

The `checkDevice()` method is simple enough, simply throwing an `UnsupportedOperationException` if the device is not running at least API Level 8:

```
  public static void checkDevice(Context context) {
    int version=Build.VERSION.SDK_INT;
    if (version < 8) {
      throw new UnsupportedOperationException(
                                              "Device must be at least "
                                                  + "API Level 8 (instead of "
                                                  + version + ")");
    }
```

The `checkManifest()` method makes heavy use of `PackageManager` to validate:

- That you have defined the custom permission
- That there is a receiver listening for GCM broadcasts and requiring the GCM permission

```
  public static void checkManifest(Context context) {
    PackageManager packageManager=context.getPackageManager();
    String packageName=context.getPackageName();
    String permissionName=packageName + ".permission.C2D_MESSAGE";
    // check permission
    try {
      packageManager.getPermissionInfo(permissionName,
                                       PackageManager.GET_PERMISSIONS);
    }
    catch (NameNotFoundException e) {
      throw new IllegalStateException(
                                      "Application does not define permission "
                                          + permissionName);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
    // check receivers
    PackageInfo receiversInfo;
    try {
      receiversInfo=
          packageManager.getPackageInfo(packageName,
                                        PackageManager.GET_RECEIVERS);
    }
    catch (NameNotFoundException e) {
      throw new IllegalStateException(
                                      "Could not get receivers for package "
                                         + packageName);
    }
    ActivityInfo[] receivers=receiversInfo.receivers;
    if (receivers == null || receivers.length == 0) {
      throw new IllegalStateException("No receiver for package "
          + packageName);
    }
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
      Log.v(TAG, "number of receivers for " + packageName + ": "
          + receivers.length);
    }
    Set<String> allowedReceivers=new HashSet<String>();
    for (ActivityInfo receiver : receivers) {
      if (PERMISSION_GCM_INTENTS.equals(receiver.permission)) {
        allowedReceivers.add(receiver.name);
      }
    }
    if (allowedReceivers.isEmpty()) {
      throw new IllegalStateException("No receiver allowed to receive "
          + PERMISSION_GCM_INTENTS);
    }
    checkReceiver(context, allowedReceivers, INTENT_FROM_GCM_MESSAGE);
  }

  private static void checkReceiver(Context context,
                                    Set<String> allowedReceivers,
                                    String action) {
    PackageManager pm=context.getPackageManager();
    String packageName=context.getPackageName();
    Intent intent=new Intent(action);
    intent.setPackage(packageName);
    List<ResolveInfo> receivers=
        pm.queryBroadcastReceivers(intent,
                                   PackageManager.GET_INTENT_FILTERS);
    if (receivers.isEmpty()) {
      throw new IllegalStateException("No receivers for action "
          + action);
    }
    if (Log.isLoggable(TAG, Log.VERBOSE)) {
      Log.v(TAG, "Found " + receivers.size() + " receivers for action "
          + action);
    }
    // make sure receivers match
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    for (ResolveInfo receiver : receivers) {
      String name=receiver.activityInfo.name;
      if (!allowedReceivers.contains(name)) {
        throw new IllegalStateException("Receiver " + name
            + " is not set with permission " + PERMISSION_GCM_INTENTS);
      }
    }
```

`GCMRegistrarCompat` offers three public methods for working with the registration ID:

- `getRegistrationId()` will return the registration ID saved in `SharedPreferences`, if it exists, if the app version is the same as before, and if the registration is not expired
- `clearRegistrationId()` simply wipes out the saved registration ID, if there is one
- `setRegistrationId()` updates `SharedPreferences` with a newly-retrieved registration ID, also saving the app version

```
  public static String getRegistrationId(Context context) {
    final SharedPreferences prefs=getGCMPreferences(context);
    String registrationId=prefs.getString(PROPERTY_REG_ID, "");
    // check if app was updated; if so, it must clear
    // registration id to
    // avoid a race condition if GCM sends a message
    int oldVersion=
        prefs.getInt(PROPERTY_APP_VERSION, Integer.MIN_VALUE);
    int newVersion=getAppVersion(context);
    if (oldVersion != Integer.MIN_VALUE && oldVersion != newVersion) {
      Log.v(TAG, "App version changed from " + oldVersion + " to "
          + newVersion + "; resetting registration id");
      clearRegistrationId(context);
      registrationId="";
    }
    else if (isRegistrationExpired(context)) {
      Log.v(TAG, "Registration expired; resetting registration id");
      clearRegistrationId(context);
      registrationId="";
    }
    return registrationId;
  }

  public static String clearRegistrationId(Context context) {
    return setRegistrationId(context, "");
  }

  private static String setRegistrationId(Context context, String regId) {
    final SharedPreferences prefs=getGCMPreferences(context);
    String oldRegistrationId=prefs.getString(PROPERTY_REG_ID, "");
    int appVersion=getAppVersion(context);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    long expirationTime=
        System.currentTimeMillis() + REGISTRATION_EXPIRY_TIME_MS;
    Editor editor=prefs.edit();

    editor.putString(PROPERTY_REG_ID, regId);
    editor.putInt(PROPERTY_APP_VERSION, appVersion);
    editor.putLong(PROPERTY_ON_SERVER_EXPIRATION_TIME, expirationTime);
    editor.commit();

    return oldRegistrationId;
  }

  private static int getAppVersion(Context context) {
    try {
      PackageInfo packageInfo=
          context.getPackageManager()
                 .getPackageInfo(context.getPackageName(), 0);
      return packageInfo.versionCode;
    }
    catch (NameNotFoundException e) {
      // should never happen
      throw new RuntimeException("Coult not get package name: " + e);
    }
  }

  private static boolean isRegistrationExpired(Context context) {
    final SharedPreferences prefs=getGCMPreferences(context);
    // checks if the information is not stale
    long expirationTime=
        prefs.getLong(PROPERTY_ON_SERVER_EXPIRATION_TIME, -1);
    return System.currentTimeMillis() > expirationTime;
  }

  private static SharedPreferences getGCMPreferences(Context context) {
    return context.getSharedPreferences(PREFERENCES,
                                        Context.MODE_PRIVATE);
  }
```

`GCMRegistrarCompat` also has a public static inner class, `BaseRegisterTask`, that handles calling `register()` on a `GoogleCloudMessaging` instance on a background thread, calling `setRegistrationId()` to save the resulting value for later use, plus calling `sendRegistrationIdToServer()` on that same background thread:

```
  static public class BaseRegisterTask extends
      AsyncTask<String, Void, String> {
    protected Context context=null;

    BaseRegisterTask(Context context) {
      this.context=context;
    }

    @Override
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    protected String doInBackground(String... params) {
      GoogleCloudMessaging gcm=
          GoogleCloudMessaging.getInstance(context);
      String regid=null;

      try {
        regid=gcm.register(params[0]);
        setRegistrationId(context, regid);
        sendRegistrationIdToServer(regid);
      }
      catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }

      return(regid);
    }

    protected void sendRegistrationIdToServer(String regid) {
      // no-op -- subclasses should override and send
      // registration id to server by some means
    }

    // override this to do something more, note that it
    // is called on a background thread!

    protected void onError(IOException e) {
      Log.e(getClass().getSimpleName(),
            "Exception registering for GCM", e);
    }
  }
```

You can elect to override `sendRegistrationIdToServer()` to actually send the registration ID to your Web service or other source of push messages, by whatever means you feel is appropriate. As it is called on a background thread, you will be able to do network I/O without freezing the UI. There is also an `onError()` method that you can elect to override, if desired, to handle any unexpected `IOExceptions` that may be raised while trying to contact the GCM server.

GCMBroadcastReceiverCompat

The `BroadcastReceiver` that we have receiving GCM broadcasts, named `GCMBroadcastReceiverCompat`, serves in a role similar to `GCMBroadcastReceiver` from the original GCM client JAR. It forwards the received broadcasts along to a service for processing, so that work can more easily be done in a background thread.

Since we have no guarantee that the device is going to stay awake very long, this is a fine use case for the `WakefulIntentService` profiled earlier in the book:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.gcm.client;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class GCMBroadcastReceiverCompat extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    intent.setClass(context, GCMIntentService.class);

    WakefulIntentService.sendWakefulWork(context, intent);
  }
}
```

To get the broadcasted `Intent` over to the service, we simply call `setClass()` on the `Intent`, identifying our service (`GCMIntentService`). This way, it does not matter what action is on the `Intent`, as we are specifying the component that this `Intent` should be delivered to.

If your app is going to do something fairly cheap when a message comes in, though, you are welcome to just do it in the `BroadcastReceiver` itself. Here, "fairly cheap" means "can be done in just a few milliseconds", and so disk or network I/O should be delegated to a service with a background thread.

GCMBaseIntentServiceCompat

The deprecated GCM client JAR offered a `GCMBaseIntentService` class that you could extend, where `GCMBaseIntentService` would call dedicated methods on your subclass, like `onMessage()`, for the different sorts of events that could arrive via the forwarded broadcast. `GCMBaseIntentServiceCompat` is a `WakefulIntentService` that offers similar behavior:

```
package com.commonsware.android.gcm.client;

import android.content.Intent;
import com.commonsware.cwac.wakeful.WakefulIntentService;
import com.google.android.gms.gcm.GoogleCloudMessaging;

abstract public class GCMBaseIntentServiceCompat extends
    WakefulIntentService {
  abstract protected void onMessage(Intent message);

  abstract protected void onError(Intent message);

  abstract protected void onDeleted(Intent message);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  public GCMBaseIntentServiceCompat(String name) {
    super(name);
  }

  @Override
  protected void doWakefulWork(Intent i) {
    GoogleCloudMessaging gcm=GoogleCloudMessaging.getInstance(this);
    String messageType=gcm.getMessageType(i);

    if (GoogleCloudMessaging.MESSAGE_TYPE_SEND_ERROR.equals(messageType)) {
      onError(i);
    }
    else if (GoogleCloudMessaging.MESSAGE_TYPE_DELETED.equals(messageType)) {
      onDeleted(i);
    }
    else {
      onMessage(i);
    }
  }
}
```

Inside of `doWakefulWork()`, we use a `GoogleCloudMessaging` instance to examine the message type of the received broadcast. Depending upon whether it is `MESSAGE_TYPE_SEND_ERROR`, `MESSAGE_TYPE_DELETED`, or a regular message, drives calls to the abstract `onError()`, `onDeleted()`, and `onMessage()` methods, respectively.

The Service

Our service, named `GCMIntentService`, extends `GCMBaseIntentServiceCompat` and overrides the three abstract methods (`onMessage()`, `onError()`, `onDeleted()`):

```
package com.commonsware.android.gcm.client;

import android.content.Intent;
import android.os.Bundle;
import android.util.Log;

public class GCMIntentService extends GCMBaseIntentServiceCompat {
  public GCMIntentService() {
    super("GCMIntentService");
  }

  @Override
  protected void onMessage(Intent message) {
    dumpEvent("onMessage", message);
  }

  @Override
  protected void onError(Intent message) {
    dumpEvent("onError", message);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  }

  @Override
  protected void onDeleted(Intent message) {
    dumpEvent("onDeleted", message);
  }

  private void dumpEvent(String event, Intent message) {
    Bundle extras=message.getExtras();

    for (String key : extras.keySet()) {
      Log.d(getClass().getSimpleName(),
            String.format("%s: %s=%s", event, key,
                          extras.getString(key)));
    }
  }
}
```

Each simply passes the supplied `Intent` along to a `dumpEvent()` private method that iterates over all of the extras and writes them out to LogCat. This way, when we send a push message, we will be able to see in LogCat what information makes it over to the service.

The "Server"

Our command-line Java app for sending messages can be found in the Push/GCMCommand sample project. Note that this is a plain Java project, not an Android project, as this is designed to run on your desktop or server, not on your device. Also note that four total JARs are needed to run this app:

1. The project's own JAR
2. `gcm-server.jar`, from the GCM portion of your SDK's `extras/` area
3. `json-simple-1.1.jar`, a compatible implementation of the `json-simple` library used by the GCM JAR
4. `commons-cli-1.2.jar`, which is used for command-line argument processing for this app

From a GCM standpoint, the `GCM` class in this sample project has a static `sendMessage()` method that does the work of sending a message to one or more devices:

```
  private static void sendMessage(String apiKey, List<String> devices,
                                  Properties data) throws Exception {
    Sender sender=new Sender(apiKey);
    Message.Builder builder=new Message.Builder();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    for (Object o : data.keySet()) {
      String key=o.toString();

      builder.addData(key, data.getProperty(key));
    }

    MulticastResult mcResult=sender.send(builder.build(), devices, 5);

    for (int i=0; i < mcResult.getTotal(); i++) {
      Result result=mcResult.getResults().get(i);

      if (result.getMessageId() != null) {
        String canonicalRegId=result.getCanonicalRegistrationId();

        if (canonicalRegId != null) {
          System.err.println(String.format("%s canonical ID = %s",
                                           devices.get(i),
                                           canonicalRegId));
        }
        else {
          System.out.println(String.format("%s success", devices.get(i)));
        }
      }
      else {
        String error=result.getErrorCodeName();

        if (Constants.ERROR_NOT_REGISTERED.equals(error)) {
          System.err.println(String.format("%s is unregistered",
                                           devices.get(i)));
        }
        else if (error != null) {
          System.err.println(String.format("%s error = %s",
                                           devices.get(i), error));
        }
      }
    }
  }
```

The pieces of information we need to send a message are:

- Our GCM API key
- A list of the device registration IDs to which GCM should deliver our message
- The key/value pairs of data that form our message payload

The Apache Commons CLI logic in this class' static `main()` method will extract these values from the command line and give them to `sendMessage()` for processing.

To send the message, `sendMessage()` first creates a GCM `Sender` object, supplying our API key to the constructor. Then, it creates a `Message.Builder`, which is a

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

builder for constructing a GCM message to be sent by that `Sender`. We use `addData()` on the builder to attach our key/value pairs, extracted from the `java.util.Properties` object that Apache Commons CLI used to give us the key/value pairs specified on the command line.

Actually sending the message then is a matter of calling `send()` on the `Sender`, supplying the `Message` built by the builder, the list of device registration IDs, and the number of retries in case there are issues in delivering the message (e.g., the server farm is swamped). We get back a `MulticastResult` object, containing details of what happened for each device in our list of devices.

`MulticastResult` is really a collection of individual `Result` objects, one per device. Each `Result` will tell us what happened, with four major possibilities:

1. Our request to enqueue the message succeeded, and the device should receive it momentarily if it is online and connected to Google.
2. Our request succeeded, but Google would like us to use a different registration ID in the future. We are given the revised ID via `getCanonicalRegistrationId()`. In a production app, we would have some sort of database of registered devices; we would replace the old registration ID with the new canonical one in that database when this result occurs.
3. Our request failed, because the device is no longer registered (e.g., the app was uninstalled). In a production app, we should remove this device from our database.
4. Our request failed for some other reason (e.g., our request was attacked by ninjas). We get an error message as a string indicating the reason for the failure; a production app would log this somewhere.

In the case of this sample command-line client, these are simply logged to `stdout` or `stderr`.

The sample project has a Linux shell script that wraps up building Java command line:

```
java -cp libs/commons-cli-1.2.jar:libs/gcm-server.jar:libs/
json_simple-1.1.jar:dist/gcm-cmd.jar \
com.commonsware.android.gcm.cmd.GCM "$@"
```

(note: once again, the trailing `\` on the first line indicates that this should be all on one line)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To use the script, switch to the `GCMCommand` directory, run `./gcm`, supplying the following command-line switches:

- `-a` or `--apiKey` with the value of your API key (note: **not** your sender ID!)
- `-d` or `--device` with the value of a device's registration ID (can have one or several of these switches)
- `-D` or `--data`, with the `key=value` pair of some data to send to a device (can have one or several of these switches)

For example, this command would send `foo=bar` to a device:

```
./gcm -a your-api-key-here -D foo=bar -d your-device-reg-id-here
```

The results will be printed to standard output, one line per `-d` switch, in the order of the `-d` switches (e.g., first registration ID in the command maps to the first line in the output).

Message Options and Advanced Features

Most of what you get out of GCM comes just from the code we have implemented so far. However, there are other things you can configure with your message to tailor delivery behavior. These come in the form of special key/value pairs added to the message payload itself, serving in the role of metadata. Note that since they elected to blend the metadata in with your data, you cannot yourself have data with the same keys as are used by Google for metadata. And, unfortunately, they did not elect to namespace their keys — even with something as trivial as a leading underscore – to help prevent collisions.

Collapse Keys

Google is not considering GCM to be a guaranteed store-and-forward queue system. In particular, Google reserves the right to try to coalesce messages, in part to reduce storage demands, but also so as not to flood the device when a connection is re-established.

Key to this is the `collapse_key` key on the message request. If a device is unavailable, and during that time you send two or more messages with the same `collapse_key`, the GCM servers may elect to only send one of those messages — typically the last one, though not necessarily. You can use this to your advantage, to minimize processing you need to do on the client. For example, if your use of GCM

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

is to alert your custom email application that "you've got mail", you can use a consistent `collapse_key` with messages telling the client how many unread emails are in their inbox. That can be used by the client to update a `Notification` and, eventually, cajole the user into actually reading her mail. In this case, you do not need the device to receive two messages in a short timespan to raise this `Notification`, so collapsing those into a single message is good for everyone.

For example, using the `gcm` script, `-D collapse_key=inbox` would set the `collapse_key` of the request to `inbox`, coalescing it with any other messages that you have sent to this device, with the same `collapse_key`, that have not yet been delivered.

A related optional parameter you can include in your messages is `delay_while_idle`. If you specify this key with a value of `true`, that will indicate to the GCM servers that, while you want the message to be delivered, it is not important enough to wake up the device. GCM will hold onto the message (or the last one if several are sent with the same `collapse_key`), but it will not push it to the device until it knows the device is awake (perhaps due to another GCM message for that device lacking this parameter). You can think of this as being akin to choosing an `AlarmManager` alarm type lacking the `_WAKEUP` suffix. The goal is to minimize battery consumption.

Note that if you eschew `collapse_key`, and you send a lot of messages without the device receiving them (e.g., it is powered down), you will run into problems. Right now, there is a limit of 100 queued messages. If you hit that limit, all queued messages are dumped, and the device will be sent "a special message indicating that the limit was reached". It will be up to your app to handle this scenario, typically by assuming that your data from the server is very stale and needs to be completely reloaded. Note that the structure of this "special message" is, alas, undocumented.

The `Message.Builder` in the GCM server JAR has dedicated `collapseKey()` and `delayWhileIdle()` methods to set these values.

Time-To-Live

If you are using `collapse_key`, you can also control how long the message will remain cached on the server, via a `time_to_live` value specified in seconds. The default is four weeks. But if you know the message is useless after a shorter period of time (e.g., after some real-world event has occurred), specifying a `time_to_live` can purge this message and prevent the app on the device from displaying something useless to the user.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `Message.Builder` in the GCM server JAR has a dedicated `timeToLive()` method to set this value.

Re-Registration

In an ideal world, we would have our apps generate a registration ID once and be done with it for any given device. However, there are scenarios in which our app will need to re-register and supply a revised registration ID to the server:

- if the user uninstalls and reinstalls our app
- if the user taps the "Clear Data" button in the Settings app, to wipe out our app's internal storage (where the registration ID is held)
- when we upgrade our app (i.e., ship an app with a different `android:versionCode` in the manifest)

While the first two might not be terribly surprising, the latter one might be. The source code for `GCMRegistrar`, as of the time of this writing, has the following comment in `getRegistrationId()`:

```
// check if app was updated; if so, it must clear registration id to
// avoid a race condition if GCM sends a message
```

`GCMRegistrar` keeps our app's `versionCode` in the same custom `SharedPreferences` that it uses for the registration ID. Whenever registration occurs, it saves our `versionCode`. But, the next time we go to retrieve the registration ID, if our `versionCode` is different than the one that was saved, GCM clears our saved registration ID, forcing us to get a new one.

Since this behavior seems to be undocumented, it is possible that in the future they will find some other solution and perhaps reduce the number of re-registrations that may be required.

Additional Features

There are a few other options available to you with GCM that go beyond basic push messaging.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Cloud Connection Services

"Cloud Connection Services" (CCS) is Google's branding for an XMPP connection from your server to Google's GCM server farm, replacing (or operating alongside) the classic HTTP/REST interface for sending messages.

Partly, the XMPP support is there for performance. Using HTTP as a protocol results in blocking I/O (send a message, wait to find out if it was sent), typically on a new socket each time. XMPP, unlike HTTP, uses a long-lived socket connection to the server, avoiding the socket churn. And XMPP is itself a messaging interface, originally designed for use with instant messaging clients, and so it specializes in asynchronous delivery. Hence, your server can send messages as fast as it wants along a single socket, finding out asynchronously the status of those messages.

Partly, the XMPP support is used as part of the mechanism behind upstream messages, described a bit later in this chapter.

At this time, Google is not providing any particular client library to assist with GCM over CCS, saying that "you can use most XMPP libraries to interact with CCS" without specifics.

You can learn more about CCS by visiting the CCS page in the GCM documentation.

Upstream Messages

If you are using CCS, you can implement support for upstream messages: "push" messages originating with the client, delivered to your server.

You might wonder: why bother?

The big reason is battery. Upstream messages are delivered asynchronously, as part of existing GCM operations. There are no new sockets, and GCM takes steps to make the most efficient use of the mobile radio. For non-real-time delivery of information via a one-way (device->server) channel, upstream messages has its benefits.

The `GoogleCloudMessaging` object has a `send()` method where you can indicate what server to send messages to (appending `@gcm.googleapis.com` to the server's sender ID), the locally-unique ID for the message, and the message itself, in the form of a `Bundle`. The server, in turn, will receive XMPP messages containing a JSON

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

payload with the key/value pairs from the `Bundle`, plus metadata (sending app's package and registration ID, plus the ID value included in the `send()` call).

User Notifications

Upstream messaging, in turn, enables support for what Google refers to as "user notifications".

As tablets become more popular, it is increasingly likely that a user of your app will have that app on multiple devices, such as both a phone and a tablet. A sufficiently-savvy server could track multiple GCM app registration IDs per user account and then send GCM messages to *all* the user's devices, so no matter which one happens to be in use at the moment, the message is delivered to it.

Many times, the end UI result of the GCM message is a `Notification`, to let the user know about some event (e.g., "we updated your records"). However, if you are sending the GCM message to all the user's devices, you would show the `Notification` on all those devices as well. By default, the user would need to manually dismiss that `Notification` from all those devices.

Once again, a sufficiently-savvy server and app could deal with this, by telling the server that the user dismissed the `Notification` from a certain device, and the server sending follow-on messages to the other devices that cause the app to dismiss the `Notification` programmatically.

GCM's "user notifications" attempts to simplify this somewhat.

The server can create a "registration key" representing the collection of devices for a user, then send GCM messages to that key instead of to the individual registration IDs. GCM automatically will fan out that message and deliver it to all of the devices.

Furthermore, a device can send an *upstream* message to that registration key, which will then be delivered by GCM to all of the other devices associated with that key. An app could use this to notify copies of that app on the user's other devices that a `Notification` was dismissed, so the app can dismiss the `Notification` on those other devices. The server does not need to get involved.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Considering Encryption

GCM uses encryption over the air. This includes both your server communicating with Google's server via HTTPS and Google's server communicating with your device.

However, there is still one party who has access to that data besides you and your user: Google. Google also knows the identity of your users, since their devices are the ones registering for GCM messages, and therefore those messages can be traced back to their devices. The fact that Android 4.1 and beyond eschew the need for a Google account to use GCM will help privacy somewhat.

Here are two ways of dealing with this, beyond ignoring the issue:

1. Encrypt your payload. Since GCM is expecting key/value pairs, that means that you would either encrypt each value (and their keys, if the keys might somehow be leaking data), or creating your own encrypted payload, encoding it in Base64, and using that as a single value in your GCM message. This, of course, implies that the client will be capable of decrypting your messages.
2. Have your payload be a URL pointing to some other resource that your client can access but not Google (at least not readily). In this mode, you are using GCM purely as a "tickle" to tell the client to go download some data via another secured means earlier than it might ordinarily do such a download (e.g., via a daily poll).

Issues with GCM

GCM, of course, is not perfect. As with its C2DM predecessor, it has a variety of issues, many of which will not be a problem for you, though some might be more troublesome.

Requires Play Services Framework

GCM only works on devices that have the Play Services Framework. For all intents and purposes, this means it only works on devices that have the Play Store. While most Android devices do have the Play Store, some notable ones do not, including the Kindle Fire.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you are planning to distribute your app to devices by means other than the Play Store, you will need to consider a fallback plan.

Requires API Level 8

GCM only works on devices with API Level 8 or higher — the point in time when the Google Services Framework was formally established and C2DM was added to the ecosystem.

Since the vast majority of Android devices run API Level 8 or higher, this should only be an issue for developers specifically aiming to support older devices (e.g., still have significant customers for the older app).

No SLA

Google does not offer a service level agreement (SLA) for GCM. In other words, what they offer is a "best efforts" service, but if they fail to deliver your messages for one reason or another, you have no legal recourse.

For many situations, this is less a legal problem and more of a usage problem. The fact that GCM can fail means that, from time to time, it likely *will* fail. Apps should not rely upon GCM as their sole means of getting data from a server. Instead, use GCM as an optimization, to get data faster than some slow poll (e.g., once every 24 hours) would accomplish. That way, even if GCM hiccups and loses a message or two, the data will still make it down to your users, albeit not as quickly.

Applications that need some sort of guaranteed delivery will need to seek some alternative solution where the provider offers an SLA.

4K Message Limit

C2DM had a 1K message payload limit. GCM raises that to 4K. This is still relatively small. The theory behind the increase is that it is more likely that everything you need to hand the client would be included in the GCM message versus needing additional network I/O. However, due to the fact that Google has access to GCM messages, you might need additional network I/O for privacy reasons, regardless of message size.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Amazon Simple Notification Service and GCM

Amazon has had the Simple Notification Service (SNS) as part of their Amazon Web Services (AWS) portfolio for some time. SNS has long had support for sending notification messages to SMS, email, Amazon's own Simple Queue Service (SQS), or some form of HTTP endpoint (e.g., for Web hook implementations).

In the summer of 2013, Amazon extended SNS to support GCM, as well as Apple's equivalent push notification service, so that you can use a single Web service API to deliver notifications via any of these means. This competes with other services (e.g., PushIO, PushWoosh) for cross-platform mobile push options.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Basic Use of Sensors

"Sensors" is Android's overall term for ways that Android can detect elements of the physical world around it, from magnetic flux to the movement of the device. Not all devices will have all possible sensors, and other sensors are likely to be added over time. In this chapter, we will explore the general concept of Android sensors and how to receive data from them.

Note, however, that this chapter will not get into details of detecting movement via the accelerometer, etc.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on threads. Having experience with other system-service-and-listener patterns, such as fetching locations with `LocationManager`, is helpful but not strictly required.

The Sensor Abstraction Model

When fetching locations from `LocationManager`, you do not have dedicated APIs per location-finding technology (e.g., GPS vs. WiFi hotspot proximity vs. cell-tower triangulation vs. ...). Instead, you work with a `LocationManager` system service, asking for locations using a single API, where location technologies are identified by name (e.g., `GPS_PROVIDER`).

Similarly, when working with sensors, you do not have dedicated APIs to get sensor readings from each sensor. Instead, you work with a `SensorManager` system service,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

asking for sensor events using a single API, where sensors are identified by name (e.g., `TYPE_LINEAR_ACCELERATION`).

Note, though, that there are some dedicated methods on `SensorManager` to help you *interpret* some of the sensors, particularly the accelerometer. However, those are merely helper methods; getting at the actual accelerometer data uses the same APIs that you would use to, say, access the barometer for atmospheric pressure.

Considering Rates

Usually, when working with sensors, you want to find out about changes in the sensor reading over a period of time. For example, in a driving game, where the user holds their device like a steering wheel and uses it to "turn" their virtual car, you need to know information about acceleration and positioning so long as game play is going on.

Hence, when you request a feed of sensor readings from `SensorManager`, you will specify a desired rate at which you should receive those readings. You do that by specifying an amount of delay in between readings; Android will drop sensor readings that arrive before the delay period has elapsed.

There are four standard delay periods, defined as constants on the `SensorManager` class:

1. `SENSOR_DELAY_NORMAL`, which is what most apps would use for broad changes, such as detecting a screen rotating from portrait to landscape
2. `SENSOR_DELAY_UI`, for non-game cases where you want to update the UI continuously based upon sensor readings
3. `SENSOR_DELAY_GAME`, which is faster (less delay) than `SENSOR_DELAY_UI`, to try to drive a higher frame rate
4. `SENSOR_DELAY_FASTEST`, which is the "firehose" of sensor readings, without delay

The more sensor readings you get, the faster your code has to be for using those readings, lest you take too long and starve your thread of time to do anything else. This is particularly important given that you receive these sensor events on the main application thread, and therefore the time you spend processing these events is time unavailable for screen updates. Hence, choose the slowest rate that you can that will give you acceptable granularity of output.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Reading Sensors

Sensors are event-driven. You cannot ask Android for the value of a sensor at a point in time. Rather, you register a listener for a sensor, then process the sensor events as they come in. You can unregister the listener when you are done, either because you have the reading that you need, or the user has done something (like move to another activity) that indicates that you no longer need the sensor events.

To demonstrate this, we will examine the `Sensor/Monitor` sample application, which will list all of the available sensors, plus show the incoming readings from a selected sensor.

Obtaining a SensorManager

The gateway to the sensor roster on the device is the `SensorManager` system service. You obtain one of these by calling `getSystemService()` on any `Context`, asking for the `SENSOR_SERVICE`, and casting the result to be a `SensorManager`, as seen in the `onCreate()` method of our `MainActivity`:

```
    mgr=(SensorManager)getSystemService(Context.SENSOR_SERVICE);
```

Identifying a Sensor of Interest

There are sensor types, and then there are sensors.

You might think that there would be a one-to-one mapping between these. In truth, there might be more than one sensor for a given type, the way the `SensorManager` API is set up. Regardless, somewhere along the line, you will need to identify the `Sensor` that you want to work with.

The most common pattern, if you know the type of sensor that you want, is to call `getDefaultSensor()` on `SensorManager`, supplying the type of the sensor (e.g., `TYPE_ACCELEROMETER`, `TYPE_GYROSCOPE`), where the type names are constants defined on the `Sensor` class. If there is more than one possible `Sensor` for that type, Android will give you the "default" one, which is usually a reasonable choice.

Another approach, and the one used by this sample application, is to call `getSensorList()` on `SensorManager`, which returns a `List` of all `Sensor` objects available on this device. The sample's `MainActivity` has a `getSensorList()` that returns this list, after a bit of manipulation:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@Override
public List<Sensor> getSensorList() {
  List<Sensor> unfiltered=
      new ArrayList<Sensor>(mgr.getSensorList(Sensor.TYPE_ALL));
  List<Sensor> result=new ArrayList<Sensor>();

  for (Sensor s : unfiltered) {
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.KITKAT
        || !isTriggerSensor(s)) {
      result.add(s);
    }
  }

  Collections.sort(result, new Comparator<Sensor>() {
    @Override
    public int compare(final Sensor a, final Sensor b) {
      return(a.toString().compareTo(b.toString()));
    }
  });

  return(result);
}
```

Android 4.4 started introducing some "trigger sensors", ones that are designed to deliver a single reading, then automatically become unregistered. This sample app is designed to display results from more traditional sensors that provide ongoing readings. So, `getSensorList()` calls an `isTriggerSensor()` method on API Level 19+ devices, and throws out sensors that are trigger sensors. The `isTriggerSensor()` method simply checks the sensor type against a list of trigger sensors:

```
@TargetApi(Build.VERSION_CODES.KITKAT)
private boolean isTriggerSensor(Sensor s) {
  int[] triggers=
      { Sensor.TYPE_SIGNIFICANT_MOTION, Sensor.TYPE_STEP_DETECTOR,
        Sensor.TYPE_STEP_COUNTER };

  return(Arrays.binarySearch(triggers, s.getType()) >= 0);
}
```

The reason for isolating `isTriggerSensor()` into a separate method, and not having the array of sensor types as a `static final` array, is because these sensor types are not available in all Android versions. Having the array of sensor types as a `static final` data member would require putting the `@TargetApi` annotation on the entire class, which is unwise if the class will be used on older devices. This way, we can isolate the new-target code into a dedicated method, with a more locally-scoped `@TargetApi` annotation.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Getting Sensor Events

To get sensor events, you need a `SensorEventListener`. This is an interface, calling for two method implementations:

1. `onAccuracyChanged()`, where you are informed about a significant change in the accuracy of the readings that you are going to get from the sensor
2. `onSensorChanged()`, where you are passed a `SensorEvent` representing one of those readings

To receive events for a given `Sensor`, you call `registerListener()` on the `SensorManager`, supplying the `Sensor`, the `SensorEventListener`, and one of the `SENSOR_DELAY_*` values to control the rate of events. Later on, you need to call `unregisterListener()`, supplying the same `SensorEventListener`, to break the connection. **Failing to unregister the listener is bad**. The sensor subsystem is oblivious to things like activity lifecycles, and so if you leak a listener, not only will you perhaps leak the component that registered the listener, but you will continue to get sensor events until the process is terminated. As active sensors do consume power, users will not appreciate the battery drain your leaked listener will incur.

The `List` of `Sensor` objects from that `getSensorList()` method shown previously will be used to populate a `ListView`. When the user taps on a `Sensor` in the list, an `onSensorSelected()` method is called on the `MainActivity`. Here, we unregister our listener (a `SensorLogFragment` that we will discuss more in a bit), in case we were registered for a prior `Sensor` choice, before registering for the newly-selected `Sensor`:

```
  @Override
  public void onSensorSelected(Sensor s) {
    mgr.unregisterListener(log);
    mgr.registerListener(log, s, SensorManager.SENSOR_DELAY_NORMAL);
    log.init(isXYZ(s));
    panes.closePane();
  }
```

We will discuss the remainder of the `onSensorSelected()` method a bit later in this chapter.

Since `SensorLogFragment` implements `SensorEventListener` — so we can use it with `registerListener()` — we need to implement `onAccuracyChanged()` and `onSensorChanged()`:

```
  @Override
  public void onAccuracyChanged(Sensor sensor, int accuracy) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    // unused
  }

  @Override
  public void onSensorChanged(SensorEvent e) {
    Float[] values=new Float[3];

    values[0]=e.values[0];
    values[1]=e.values[1];
    values[2]=e.values[2];

    adapter.add(values);
  }
```

Once again, we will get into the implementation of `onSensorChanged()` a bit later in this chapter.

The big thing to note now about `onSensorChanged()`, though, is that **the `SensorEvent` object comes from an object pool and gets recycled**. It is not safe for you to hold onto this `SensorEvent` object past the call to `onSensorChanged()`. Hence, you need to do something with the data in the `SensorEvent`, then let go of the `SensorEvent` itself, so that instance can be used again later. This is to help prevent excessive garbage collection, particularly for low-delay requests for sensor readings (e.g., `SENSOR_DELAY_FASTEST`).

Interpreting Sensor Events

The key piece of data in the `SensorEvent` object is `values`. This is a six-element `float` array containing the actual sensor reading. What those values *mean* will vary by sensor. For example:

- For accelerometer readings (e.g., `TYPE_ACCELEROMETER`), the first three elements of the array represent the reported acceleration, in m/s², along the X, Y, and Z axes respectively (X = out the right side of the device, Y = out the top edge of the device, Z = out the screen towards the user's eyes)
- `TYPE_PRESSURE` uses the first element of the `values` array to report the barometric pressure in millibars
- `TYPE_LIGHT` uses the first element of the `values` array to report the light level in lux

And so on.

The `SensorEvent` documentation contains instructions on how to interpret these events on a per-sensor-type basis.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That being said, sensors can be roughly divided into two groups:

1. Sensors whose readings take into account three axes (X/Y/Z). These include `TYPE_ACCELEROMETER`, `TYPE_GRAVITY`, `TYPE_GYROSCOPE`, `TYPE_LINEAR_ACCELERATION`, and `TYPE_MAGNETIC_FIELD`.
2. Sensors that have simple single-value readings, such as `TYPE_PRESSURE` and `TYPE_LIGHT`

The `isXYZ()` method on `MainActivity` simply returns a `boolean` indicating whether or not this particular `Sensor` is one that uses all three axes (`true`) or not (`false`). As the roster of sensors has changed over the years, it also does some checks based on API level:

```
  @TargetApi(Build.VERSION_CODES.KITKAT)
  private boolean isXYZ(Sensor s) {
    switch (s.getType()) {
      case Sensor.TYPE_ACCELEROMETER:
      case Sensor.TYPE_GRAVITY:
      case Sensor.TYPE_GYROSCOPE:
      case Sensor.TYPE_LINEAR_ACCELERATION:
      case Sensor.TYPE_MAGNETIC_FIELD:
      case Sensor.TYPE_ROTATION_VECTOR:
        return(true);
    }

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR2) {
      if (s.getType() == Sensor.TYPE_GAME_ROTATION_VECTOR
          || s.getType() == Sensor.TYPE_GYROSCOPE_UNCALIBRATED
          || s.getType() == Sensor.TYPE_MAGNETIC_FIELD_UNCALIBRATED) {
        return(true);
      }
    }

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
      if (s.getType() == Sensor.TYPE_GEOMAGNETIC_ROTATION_VECTOR) {
        return(true);
      }
    }

    return(false);
  }
```

Wiring Together the Sample

Overall, this sample app uses the `SlidingPaneLayout` first seen back in the chapter on large-screen support. We have two fragments, in a master-detail pattern, where the "master" will be a list of all available sensors, and the "detail" will be a log of sensor readings from a selected sensor.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Our layout (`res/layout/activity_main.xml`) wires in a `SensorsFragment` (master) and `SensorLogFragment` (detail) in a `SlidingPaneLayout`:

```
<android.support.v4.widget.SlidingPaneLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/panes"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <fragment
    android:id="@+id/sensors"
    android:name="com.commonsware.android.sensor.monitor.SensorsFragment"
    android:layout_width="300sp"
    android:layout_height="match_parent"/>

  <fragment
    android:id="@+id/log"
    android:name="com.commonsware.android.sensor.monitor.SensorLogFragment"
    android:layout_width="400dp"
    android:layout_height="match_parent"
    android:layout_weight="1"/>

</android.support.v4.widget.SlidingPaneLayout>
```

The `SensorsFragment` is reminiscent of `CountriesFragment` from the `SlidingPaneLayout` variant of the `EU4You` sample. The biggest differences are that we use a `SensorListAdapter` for representing the list of sensors, that we use `getSensorList()` on our `SensorsFragment.Contract` class to retrieve the model data, and that we call `onSensorSelected()` on the contract to report of selections:

```
package com.commonsware.android.sensor.monitor;

import android.hardware.Sensor;
import android.os.Bundle;
import android.view.View;
import android.widget.ListView;
import java.util.List;

public class SensorsFragment extends
    ContractListFragment<SensorsFragment.Contract> {
  static private final String STATE_CHECKED=
      "com.commonsware.android.sensor.monitor.STATE_CHECKED";
  private SensorListAdapter adapter=null;

  @Override
  public void onActivityCreated(Bundle state) {
    super.onActivityCreated(state);

    adapter=new SensorListAdapter(this);
    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
    setListAdapter(adapter);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (state != null) {
      int position=state.getInt(STATE_CHECKED, -1);

      if (position > -1) {
        getListView().setItemChecked(position, true);
        getContract().onSensorSelected(adapter.getItem(position));
      }
    }
  }

  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    l.setItemChecked(position, true);

    getContract().onSensorSelected(adapter.getItem(position));
  }

  @Override
  public void onSaveInstanceState(Bundle state) {
    super.onSaveInstanceState(state);

    state.putInt(STATE_CHECKED, getListView().getCheckedItemPosition());
  }

  interface Contract {
    void onSensorSelected(Sensor s);

    List<Sensor> getSensorList();
  }
}
```

`SensorListAdapter` illustrates another approach for handling the difference in "activated" row support. The `EU4You` samples used an `activated` style to apply the "activated" support on Android 3.0 and higher. Here, our custom `ArrayAdapter` subclass dynamically chooses between `android.R.layout.simple_list_item_activated_1` (an activated-capable built-in row layout) and the classic `android.R.layout.simple_list_item_1` based upon API level:

```
package com.commonsware.android.sensor.monitor;

import android.hardware.Sensor;
import android.os.Build;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.TextView;

class SensorListAdapter extends ArrayAdapter<Sensor> {
  SensorListAdapter(SensorsFragment sensorsFragment) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super(sensorsFragment.getActivity(), getRowResourceId(),
          sensorsFragment.getContract().getSensorList());
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    View result=super.getView(position, convertView, parent);

    ((TextView)result).setText(getItem(position).getName());

    return(result);
  }

  private static int getRowResourceId() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD) {
      return(android.R.layout.simple_list_item_activated_1);
    }

    return(android.R.layout.simple_list_item_1);
  }
}
```

We also have to override `getView()`, as our model is `Sensor`, whose `toString()` is not what we want, so we have to manually populate the list row with `getName()` instead.

`SensorLogFragment` is another `ListFragment`. In particular, though, we set it up for `TRANSCRIPT_MODE_NORMAL`, which means that Android will automatically scroll the `ListView` to the bottom if we add new rows to the list *and* the user has not scrolled up in the list to view past data:

```
  @Override
  public void onActivityCreated(Bundle state) {
    super.onActivityCreated(state);

    getListView().setTranscriptMode(ListView.TRANSCRIPT_MODE_NORMAL);
  }
```

However, we do not initialize our `ListAdapter` in `onActivityCreated()`, as we might normally do. Instead, we have a dedicated `init()` method, to be called by `MainActivity`, where we set up the `SensorLogAdapter` and keep track of whether the `Sensor` that we are logging is designed to report three-dimensional values (`isXYZ` is `true`) or not:

```
  void init(boolean isXYZ) {
    this.isXYZ=isXYZ;
    adapter=new SensorLogAdapter(this);
    setListAdapter(adapter);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `init()` method, in turn, was called by `onSensorSelected()` of `MainActivity`. Hence, whenever the user taps on a sensor, we set up a fresh log. `init()` can do this because `MainActivity` retrieved our `SensorLogFragment` up in `onCreate()`, stashing it in a `log` data member:

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    mgr=(SensorManager)getSystemService(Context.SENSOR_SERVICE);
    log=
        (SensorLogFragment)getFragmentManager().findFragmentById(R.id.log);

    panes=(SlidingPaneLayout)findViewById(R.id.panes);
    panes.openPane();
  }
```

Our `onSensorChanged()` method in `SensorLogFragment` copies the `values` from the `SensorEvent` into a separate `Float` array that is our list's model data:

```
  @Override
  public void onSensorChanged(SensorEvent e) {
    Float[] values=new Float[3];

    values[0]=e.values[0];
    values[1]=e.values[1];
    values[2]=e.values[2];

    adapter.add(values);
  }
```

`SensorLogAdapter` uses the `isXYZ` value to determine how it should format the rows:

- For single-value sensors, we just show the first `Float` from the array
- For three-dimensional sensors, we show all three dimensions, plus the "net" (square root of the sum of the squares), separated by slashes

```
  class SensorLogAdapter extends ArrayAdapter<Float[]> {
    public SensorLogAdapter(SensorLogFragment sensorLogFragment) {
      super(sensorLogFragment.getActivity(),
            android.R.layout.simple_list_item_1,
            new ArrayList<Float[]>());
    }

    @SuppressLint("DefaultLocale")
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      TextView row=
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        (TextView)super.getView(position, convertView, parent);
    String content=null;
    Float[] values=getItem(position);

    if (isXYZ) {
      content=
          String.format("%7.3f / %7.3f / %7.3f / %7.3f",
                        values[0],
                        values[1],
                        values[2],
                        Math.sqrt(values[0] * values[0] + values[1]
                            * values[1] + values[2] * values[2]));
    }
    else {
      content=String.format("%7.3f", values[0]);
    }

    row.setText(content);

    return(row);
  }
}
```

The rest of `MainActivity` simply manages the `SlidingPaneLayout`, much like the `EU4YouSlidingPane` sample did.

The Results

When the user taps on a sensor in the list, we get a log of readings:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

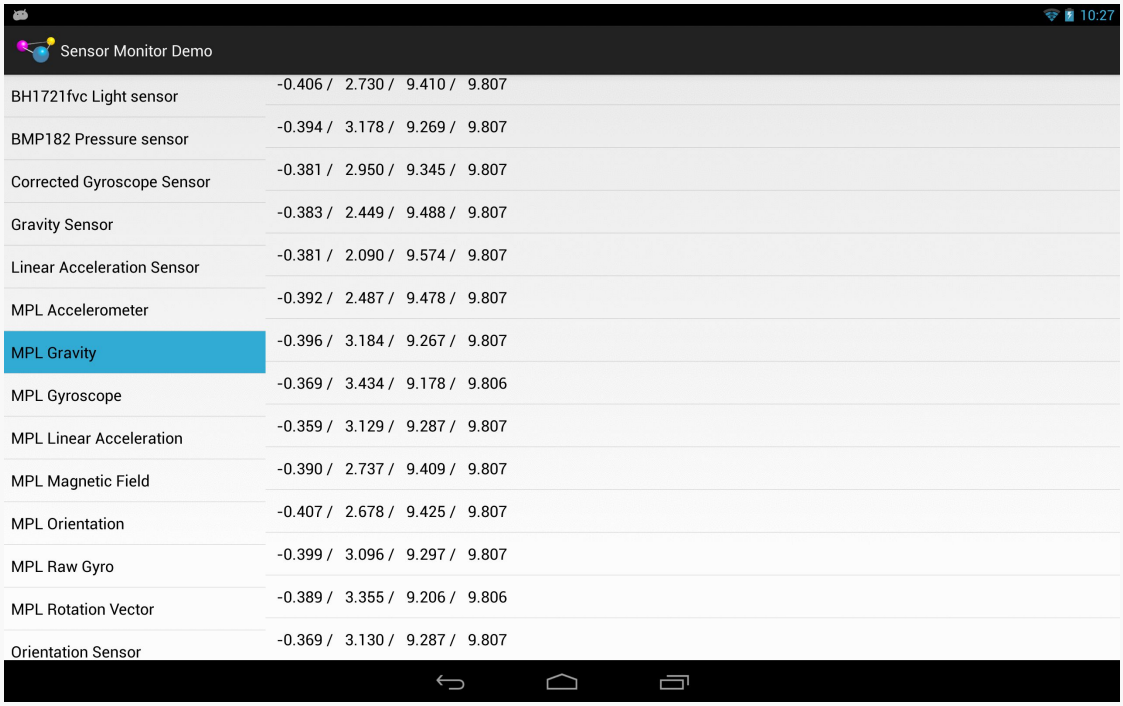


Figure 683: SensorMonitor, On a Nexus 10, Showing Gravity Readings While Being Wiggled by the Author

Batching Sensor Readings

API Level 19 (Android 4.4) added a new feature to the sensor subsystem: batched sensor events. Now, `registerListener()` can take a batch period in microseconds, and Android may elect to deliver events to you delayed by up to that amount of time. The objective will be to reduce the power draw of the sensors, for sensor hardware that supports this sort of batching behavior. Not all hardware will, in which case your requested batch latency will be ignored.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Printing and Document Generation

Mobile devices are continuing to close the gap on capabilities that had formerly been the sole province of desktop systems or servers. After all, if the vision is that people should be able to use phones and tablets *instead of* desktops and notebooks, phones and tablets need to do whatever it is that those people need to have done.

One such capability is the ability to print to networked printers. While various third-party printing options have been available for some time, it is only with the Android 4.4 release that the OS and framework itself has support for printing. In the fullness of time, a preponderance of Android devices will be natively capable of printing, and so users will be more likely to expect that your app supports such printing.

As it turns out, the print engine in Android is centered upon the PDF document format, and Android supports converting HTML into PDF, albeit on a somewhat limited basis.

The API seems simple and clean. It actually *is* simple and clean... so long as you are printing very simple contents (bitmaps or HTML). Once you get into anything more complicated than that, the threading alone starts to make things rather messy.

This chapter describes how to use the Android 4.4 print system, including how to print HTML and PDF files. It will also cover how to generate HTML and PDF files, whether for printing or for other purposes (e.g., reports to be emailed or uploaded somewhere).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Also, you should read the chapter on advanced uses of `WebView`.

The Android Print System

Writing programs that print on desktop operating systems historically has been tedious. The fine-grained control that is needed for high-quality output makes the APIs complicated, and these tend to be only partially masked by high-level wrappers to simplify common scenarios.

Android's print system is no different.

Starting with Android 4.4, you can request access to a `PrintManager` system service (via `getSystemService()`, called on any `Context`). It offers a `print()` method that lets you describe what should be printed, in the form of a `PrintAttributes` (e.g., what size paper are you looking for?) and a `PrintDocumentAdapter`. The latter is responsible for working with Android to actually create the content to be printed.

`print()` returns a `PrintJob`, which you can use to examine the status of the print request. `PrintManager` also offers a `getPrintJobs()` method that returns all of *your* outstanding print requests. Note that you cannot access print jobs from other applications.

Hence, the real complexity of printing lies in the `PrintDocumentAdapter` implementation. This class is responsible for generating a PDF that represents the content to be printed. This leads to four basic ways of working with a `PrintDocumentAdapter`:

1. Have one created for you, such as via a `WebView` for printing HTML content
2. Create one that takes a PDF generated elsewhere and uses it for the output
3. Create one that uses Android's `Canvas`-based PDF generation class, called `PrintedPdfDocument`
4. Use APIs that avoid all of this entirely, such as `printBitmap()` on `PrintHelper`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

About the Sample App

The Printing/PrintManager sample project demonstrates all but the `Canvas` option.

The UI is just a large `EditText`, designed for you to type in a message.

The action bar overflow contains four options:

- "Bitmap", to print an image from your device or emulator
- "Web Page", to print the Web page for this book
- "TPS Report", which prints a report containing the message from the `EditText`
- "PDF", which prints a copy of the cover of Version 5.8 of this book, which is packaged in the app as an asset

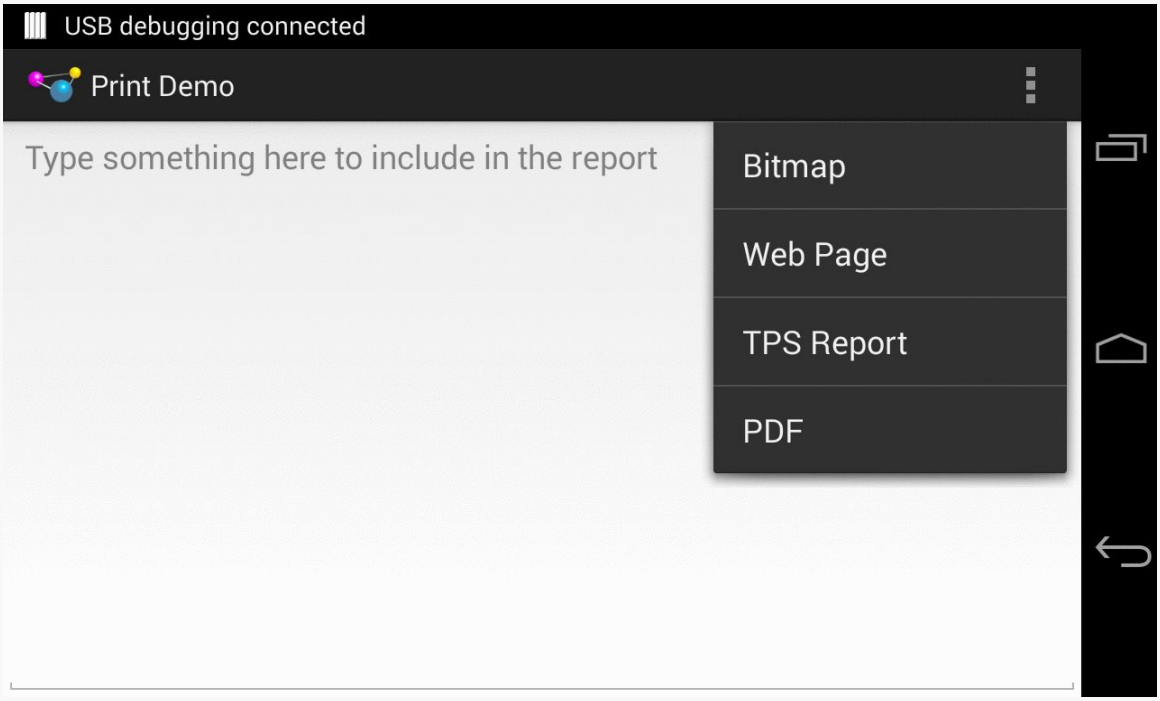


Figure 684: Print Demo App, Showing Overflow

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Printing a Bitmap

Google helpfully supplies a `PrintHelper` class in the Android Support package that makes it trivially easy to print a bitmap. Just call `printBitmap()` on the `PrintHelper`, after some minor configuration, and it takes over from there.

In `onOptionsItemSelected()` of the sample app's `MainActivity`, when the user chooses the "Bitmap" item, we call `startActivityForResult()` on an `ACTION_GET_CONTENT` `Intent`, to allow the user to pick an image from the device or emulator:

```
        Intent i=
            new Intent(Intent.ACTION_GET_CONTENT).setType("image/*");

        startActivityForResult(i, IMAGE_REQUEST_ID);
```

This, in turn, will trigger a call to `onActivityResult()`, once the user has (presumably) chosen an image:

```
  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode == IMAGE_REQUEST_ID
        && resultCode == Activity.RESULT_OK) {
      try {
        PrintHelper help=new PrintHelper(this);

        help.setScaleMode(PrintHelper.SCALE_MODE_FIT);
        help.printBitmap("Photo!", data.getData());
      }
      catch (FileNotFoundException e) {
        Log.e(getClass().getSimpleName(), "Exception printing bitmap",
              e);
      }
    }
  }
```

If the user did indeed choose an image, we create an instance of `PrintHelper`, call `setScaleMode()` to tell it fit the image to the page, and then call `printBitmap()` to print the image.

`setScaleMode()` takes one of two values:

1. `SCALE_MODE_FIT` will show the entire image, blown up as big as possible

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. `SCALE_MODE_FILL` will fill the entire page, at the cost of cropping the image along one axis, if the image's aspect ratio does not match the paper's aspect ratio

`printBitmap()` takes the name of the print job (so the user, when reviewing the outstanding print jobs, knows what it is) and either a `Uri` or a `Bitmap` for the image itself. In the case of a `Uri`, the `Uri` could be malformed, in which case the `FileNotFoundException` may be thrown, which is why we catch it.

What the user sees, after choosing an image to print (and a printer, if the user has more than one available), is a print configuration dialog appear, much like those you might see in a desktop OS:

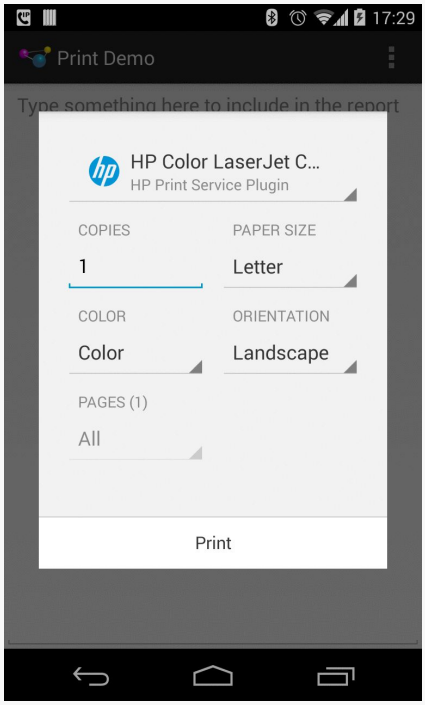


Figure 685: HP Print Configuration Dialog

The dialog itself is provided by Android; the contents of the dialog is provided by a `PrintService` that is responsible for taking our print job and actually dispatching it to the printer.

Here, the user can make typical changes, like portrait/landscape printing and the number of copies, before pressing the "Print" button. At that point, the user's chosen image will be printed.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that, in Android 4.4, the print dialog does not work especially well in landscape on smaller screen sizes, forcing the user to scroll to get to all of the widgets, including the "Print" button.

Printing an HTML Document

Printing a bitmap is nice. It is not especially useful, as it implies that we have a bitmap worth printing... and printing by itself. That is certainly possible, but it is unlikely. Even in the case where we want to print a photo, there is a very good chance that we will need to print some additional information along with the photo (caption, date when photo was taken, etc.).

Being able to print something over which we have greater control of the rendering would be more useful. The easiest way to do that is to print some HTML. Later in this chapter we will cover how to generate some dynamic HTML representing what you want to print. For the moment, though, let's focus on the printing itself.

Printing and WebView

Starting in API Level 19, `WebView` is capable of participating in the print process. You can load up a `WebView` with your desired content, then print that content.

Some apps will already be using a `WebView` as part of the UI, and that `WebView` will contain what needs to be printed. For example, a Web browser can easily add a "Print" action bar overflow item that would print the contents of the active `WebView` in the browser.

For cases where you want to print something, but you are not using the `WebView` for anything but printing, you do *not* need to add the `WebView` to the UI. You can create a `WebView` instance via its constructor, passing in your `Activity` as the `Context` required by that constructor. You can then populate that `WebView` with what needs to be printed, then print it. That is the technique that the sample application demonstrates, in part because it is likely to be the more common scenario — only so many apps use a `WebView` in the UI, and more are likely to need to print.

Printing a URL

The sample app's "Web Page" action bar overflow item is tied to an `R.id.web MenuItem`. When that is tapped by the user, `onOptionsItemSelected()` calls `printWebPage()` to print a Web page loaded from a URL:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
private void printWebPage() {
  WebView print=prepPrintWebView(getString(R.string.web_page));

  print.loadUrl("http://commonsware.com/Android");
}
```

Here, `getString(R.string.web_page)` is returning a string resource that will be used for the name of a print job. `prepPrintWebView()` returns the `WebView` that will be used for printing. `loadUrl()` is the standard `WebView` method for populating the `WebView` from a URL. Note that this causes the sample app to need the `INTERNET` permission, since we are downloading a Web page and its related assets (CSS, images) from the Internet.

You will notice that we are not actually printing anything directly in `printWebPage()`, which may seem a bit odd given the name of the method. That is because we cannot print anything until the page is loaded — after all, it is only then that we have what we want to print.

The job of `prepPrintWebView()` is to arrange to get control when the page is loaded and actually print the desired page:

```
private WebView prepPrintWebView(final String name) {
  WebView result=getWebView();

  result.setWebViewClient(new WebViewClient() {
    @Override
    public void onPageFinished(WebView view, String url) {
      print(name, view.createPrintDocumentAdapter(),
            new PrintAttributes.Builder().build());
    }
  });

  return(result);
}
```

`getWebView()` is just a lazy-initialization method, populating a `wv` data member of the activity with a `WebView`. This way, we avoid creating the `WebView` up front, as if the user does not elect to print any HTML, we do not need the `WebView`, and a `WebView` is expensive to initialize:

```
private WebView getWebView() {
  if (wv == null) {
    wv=new WebView(this);
  }

  return(wv);
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We are holding onto the WebView in a data member to ensure that it will not be garbage-collected. A WebView that is part of our UI is being strongly held by its parent in the View hierarchy, so we do not normally need to worry about this. However, in this case, we are creating a WebView dynamically and are *not* adding it to the UI, so we are responsible for holding onto it, at least as long as is needed. In this sample, we just hold onto it for the rest of the life of the activity.

Back in prepPrintWebView(), we call setWebViewClient(), to attach an anonymous inner class extending WebViewClient to the WebView. Back in the chapter introducing WebView, we saw WebViewClient in the context of shouldOverrideUrlLoading(). Another popular method to override on a WebViewClient is onPageFinished(). This is called when the HTML and related assets (CSS, images, etc.) have been loaded and rendered within the WebView. At this point, for the particular URL we are loading, it is safe to print the page.

In onPageFinished(), we call a print() method on MainActivity itself:

```
  private PrintJob print(String name, PrintDocumentAdapter adapter,
                         PrintAttributes attrs) {
    startService(new Intent(this, PrintJobMonitorService.class));

    return(mgr.print(name, adapter, attrs));
  }
```

The first line of print() calls startService() to start a PrintJobMonitorService. We will see more about why we are doing that later in this chapter. For the moment, take it on faith that this service will help ensure that our process stays around long enough for our print job to finish.

The second line of print() calls a print() method on a mgr data member. Here, mgr is a PrintManager, initialized up in onCreate() of the activity, by calling getSystemService(), asking for the PRINT_SERVICE, and casting the result to be a PrintManager.

```
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    prose=(EditText)findViewById(R.id.prose);
    mgr=(PrintManager)getSystemService(PRINT_SERVICE);
  }
```

The print() method tells the PrintManager to go print something. print() takes three parameters:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. The name of the print job, which is kept along with the print job itself in case something (e.g., the print driver) wishes to show the user a roster of print jobs. In our case, this is that string resource passed in as the `name` parameter to `prepPrintWebView()`. That parameter is marked `final`, so the call to `setWebViewClient()` will include the value of that parameter in the anonymous inner class' implementation of `onPageFinished()`.
2. A `PrintDocumentAdapter`. For the case of printing HTML, we get one of those by calling `createPrintDocumentAdapter()` on our populated `WebView`.
3. A `PrintAttributes` object, describing any particular requirements that you have for the printed output (e.g., media size, margins, color/monochrome). If you will let the user control all of that via the print dialog, an empty `PrintAttributes` is fine to use with `print()`. You typically create a `PrintAttributes` by creating a `PrintAttributes.Builder`, calling setters on the `Builder` to configure the `PrintAttributes`, and getting the resulting `PrintAttributes` via a call to `build()`.

And that's it. Android — in particular, `WebView` and its `PrintDocumentAdapter` – takes over from here and prints the Web page.

Limitations and Concerns

Alas, we do not have infinite flexibility with printing HTML from a `WebView`. Here are some limitations and potential problem areas that you will encounter:

- While you can use JavaScript in the loaded HTML, it cannot trigger the print itself using any standard DOM methods.
- Also, if your JavaScript is going to fire off some asynchronous operations, like an AJAX request, bear in mind that `onPageFinished()` does *not* take those operations into account. You will need to use `addJavascriptInterface()` to inject a Java object into the JavaScript realm, then have your asynchronous work arrange to call some method on that Java object, to signal to you that the document is ready for printing.
- Print CSS rules, like headers, footers, page numbers, landscape properties, and the like are ignored at present.
- A `WebView` can only do one print job at a time. Printing occurs asynchronously, and so you have to be careful that you do not accidentally start off a second print job while an earlier one is in process. The `print()` method returns a `PrintJob` that you can use to monitor the print job status, and this object will be covered in a bit more detail later in this chapter. You may wish to set up a `WebView` pool, where you reuse an existing `WebView` only if its associated `PrintJob` is completed, creating a new `WebView` instance if

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

there is no available `WebView` at the moment. Or, you might disable printing options in the UI until the `PrintJob` is done, so you can reuse the `WebView`. The sample app does none of this, to keep things simpler.

- Printing HTML is "an all-or-nothing affair". You cannot print a subset of the HTML, whether denoted by HTML constructs (e.g., `<div>` IDs) or by page numbers. Hence, you need to load into the `WebView` *exactly* what you want to print, no more, no less.

Also, any direct use of `PrintManager` will only work on API Level 19. You will need to ensure that you only try using it on API Level 19+ devices, using Java version guard blocks. You will also need to set your build target to at least API Level 19 to be able to reference the `PrintManager` and related classes.

Finally, while loading and printing HTML are both intrinsically asynchronous, *generating* HTML locally is not. We will discuss this issue a bit more later in this chapter.

Printing a PDF File

As will be seen in the next section, even if we "hand-roll" our printed output using a `Canvas`, the result seems to be a PDF file. Hence, you would think that the printing framework would provide convenience code to print a PDF file that we obtained by other means.

Alas, that is not the case.

The sample app contains some code demonstrating how this is possible, inspired by this Stack Overflow answer, though it may cut a few corners that Google would prefer not be cut. However, it also illustrates how to create your own `PrintDocumentHandler`, which you will need for any print job not involving a bitmap or HTML.

The PrintDocumentAdapter Protocol

We supply a `PrintDocumentAdapter` to the `print()` method on `PrintManager`. In the HTML case, we got a `PrintDocumentAdapter` from the `WebView`, and so it is Google's job to implement that adapter. Similarly, `PrintHelper` has its own internal implementation of a `PrintDocumentAdapter` that it uses for printing the bitmap.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For anything else, you need to create your own `PrintDocumentAdapter`, or find a third-party implementation that you can perhaps reuse.

`PrintDocumentAdapter`'s job is to supply the `PrintManager` with the content to be printed, in the form of a PDF file. To do that, there are four callback methods that `PrintManager` (and related classes) will call on the `PrintDocumentAdapter`:

1. `onStart()` is called first. If you are planning on using the same `PrintDocumentAdapter` instance for multiple print jobs, this would be a spot to initialize the work for a new job. Otherwise, if you were only planning on using a `PrintDocumentAdapter` instance once, you may as well just put your initialization logic in the constructor.
2. `onLayout()` is called next. Here is where you do enough work to determine what the resulting output will be later on as printing continues. In particular, if you want to provide an accurate page count, this is where you will need to perform the necessary calculations to determine that.
3. `onWrite()` is called next, asking you to write one or more PDF pages out to a supplied `ParcelFileDescriptor` (on which you can create an `OutputStream`).
4. `onFinish()` is called last, when the printing request is completed, so you can free up any necessary resources.

Introducing ThreadedPrintDocumentAdapter

All four of those callback methods are called on the main application thread. Your `onStart()` and `onFinish()` methods need to be fast enough to complete their work on that thread, and that may not be a problem. The work that `onLayout()` and `onWrite()` do may take a while, though, and so the protocol is designed to allow you to do that work on a background thread. Both methods are passed a callback object that you use to pass along the results of your work, and both are passed a `CancellationSignal` to indicate if the user cancels the print job while you are doing the work.

What `PrintDocumentAdapter` does *not* do is actually give you a thread to use.

So, the sample app contains a `ThreadedPrintDocumentAdapter` that moves the `onLayout()` and `onFinish()` work to a background thread:

```
package com.commonsware.android.print;

import android.content.Context;
import android.os.Bundle;
import android.os.CancellationSignal;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.os.ParcelFileDescriptor;
import android.print.PageRange;
import android.print.PrintAttributes;
import android.print.PrintDocumentAdapter;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

abstract class ThreadedPrintDocumentAdapter extends
    PrintDocumentAdapter {
  abstract LayoutJob buildLayoutJob(PrintAttributes oldAttributes,
                                    PrintAttributes newAttributes,
                                    CancellationSignal cancellationSignal,
                                    LayoutResultCallback callback,
                                    Bundle extras);

  abstract WriteJob buildWriteJob(PageRange[] pages,
                                  ParcelFileDescriptor destination,
                                  CancellationSignal cancellationSignal,
                                  WriteResultCallback callback,
                                  Context ctxt);

  private Context ctxt=null;
  private ExecutorService threadPool=Executors.newFixedThreadPool(1);

  ThreadedPrintDocumentAdapter(Context ctxt) {
    this.ctxt=ctxt;
  }

  @Override
  public void onLayout(PrintAttributes oldAttributes,
                       PrintAttributes newAttributes,
                       CancellationSignal cancellationSignal,
                       LayoutResultCallback callback, Bundle extras) {
    threadPool.submit(buildLayoutJob(oldAttributes, newAttributes,
                                     cancellationSignal, callback,
                                     extras));
  }

  @Override
  public void onWrite(PageRange[] pages,
                      ParcelFileDescriptor destination,
                      CancellationSignal cancellationSignal,
                      WriteResultCallback callback) {
    threadPool.submit(buildWriteJob(pages, destination,
                                    cancellationSignal, callback, ctxt));
  }

  @Override
  public void onFinish() {
    threadPool.shutdown();

    super.onFinish();
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  protected abstract static class LayoutJob implements Runnable {
    PrintAttributes oldAttributes;
    PrintAttributes newAttributes;
    CancellationSignal cancellationSignal;
    LayoutResultCallback callback;
    Bundle extras;

    LayoutJob(PrintAttributes oldAttributes,
              PrintAttributes newAttributes,
              CancellationSignal cancellationSignal,
              LayoutResultCallback callback, Bundle extras) {
      this.oldAttributes=oldAttributes;
      this.newAttributes=newAttributes;
      this.cancellationSignal=cancellationSignal;
      this.callback=callback;
      this.extras=extras;
    }
  }

  protected abstract static class WriteJob implements Runnable {
    PageRange[] pages;
    ParcelFileDescriptor destination;
    CancellationSignal cancellationSignal;
    WriteResultCallback callback;
    Context ctxt;

    WriteJob(PageRange[] pages, ParcelFileDescriptor destination,
             CancellationSignal cancellationSignal,
             WriteResultCallback callback, Context ctxt) {
      this.pages=pages;
      this.destination=destination;
      this.cancellationSignal=cancellationSignal;
      this.callback=callback;
      this.ctxt=ctxt;
    }
  }
}
```

This class uses a single-thread thread pool, managed by an `ExecutorService`. In principle, a well-written `PrintDocumentAdapter` could handle multiple print jobs in parallel — if you attempt this and are using `ThreadedPrintDocumentAdapter` for inspiration, simply increase the size of the thread pool.

The `onLayout()` and `onWrite()` methods package up their parameters (described in the next section) into job objects. Those objects implement `Runnable`, and they are then handed to the `ExecutorService` to be run on the next-available thread. `onFinish()` shuts down the `ExecutorService`, though if you wanted to use the `ThreadedPrintDocumentAdapter` for multiple print jobs, you would come up with some other logic to clean up the `ExecutorService` when you were done with *all* of the jobs.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subclasses of `ThreadedPrintDocumentAdapter` need to:

- Create subclasses of the `LayoutJob` and `WriteJob` static inner classes, implementing their respective `run()` methods, to do the work required of `onLayout()` and `onWrite()`
- Implement `buildLayoutJob()` and `buildWriteJob()` methods that return instances of those custom subclasses

(fans of dependency injection no doubt can find better solutions for wiring up a `ThreadedPrintDocumentAdapter`)

A PdfDocumentAdapter

However, we still need to actually be able to print a PDF, which `ThreadedPrintDocumentAdapter` does not do on its own. The sample app also has a `PdfDocumentAdapter`, which extends `ThreadedPrintDocumentAdapter` and demonstrates a crude way of printing a PDF through the `PrintDocumentAdapter` protocol.

`PdfDocumentAdapter` does not use `onStart()` or `onFinish()`. And, since the `onLayout()` and `onWrite()` methods are handled by `ThreadedPrintDocumentAdapter`, `PdfDocumentAdapter` does not have those either.

It does, however, have the `buildLayoutJob()` and `buildWriteJob()` methods required by `ThreadedPrintDocumentAdapter`. These return instances of a `PdfLayoutJob` and `PdfWriteJob`, respectively:

```
@Override
LayoutJob buildLayoutJob(PrintAttributes oldAttributes,
                         PrintAttributes newAttributes,
                         CancellationSignal cancellationSignal,
                         LayoutResultCallback callback, Bundle extras) {
  return(new PdfLayoutJob(oldAttributes, newAttributes,
                          cancellationSignal, callback, extras));
}

@Override
WriteJob buildWriteJob(PageRange[] pages,
                       ParcelFileDescriptor destination,
                       CancellationSignal cancellationSignal,
                       WriteResultCallback callback, Context ctxt) {
  return(new PdfWriteJob(pages, destination, cancellationSignal,
                         callback, ctxt));
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

PdfLayoutJob needs to fulfill the bulk of the onLayout() contract:

- Monitor the CancellationSignal and call onLayoutCancelled() on the supplied LayoutResultCallback if the job has been canceled
- Populate a PrintDocumentInfo object to provide metadata about the document to be printed, and pass that to onLayoutFinished() on the LayoutResultCallback

```
private static class PdfLayoutJob extends LayoutJob {
  PdfLayoutJob(PrintAttributes oldAttributes,
               PrintAttributes newAttributes,
               CancellationSignal cancellationSignal,
               LayoutResultCallback callback, Bundle extras) {
    super(oldAttributes, newAttributes, cancellationSignal, callback,
          extras);
  }

  @Override
  public void run() {
    if (cancellationSignal.isCanceled()) {
      callback.onLayoutCancelled();
    }
    else {
      PrintDocumentInfo.Builder builder=
          new PrintDocumentInfo.Builder("CHANGE ME PLEASE");

      builder.setContentType(PrintDocumentInfo.CONTENT_TYPE_DOCUMENT)
             .setPageCount(PrintDocumentInfo.PAGE_COUNT_UNKNOWN)
             .build();

      callback.onLayoutFinished(builder.build(),
                                !newAttributes.equals(oldAttributes));
    }
  }
}
```

PdfLayoutJob also has access to two PrintAttributes objects, the "old" attributes and the "new" attributes. In principle, onLayout() could be called a couple of times, perhaps based upon changes the user makes in the print dialog. These PrintAttributes objects describe the nature of the output, including things like page size and margins. PdfLayoutJob totally ignores these, because the PDF is a packaged asset in this case and cannot be changed. If you are dynamically generating a PDF file, you may wish to pay attention to the new PrintAttributes and take them into account.

PdfLayoutJob also has access to a Bundle of "extras", not unlike the "extras" associated with an Intent. At the present time, there is only one semi-documented "extra", EXTRA_PRINT_PREVIEW, which will be true if onLayout() is being called to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

generate a print preview of the printed output, `false` otherwise. There is no obvious way to trigger a print preview, from code or the print dialog, and so `PdfLayoutJob` ignores this as well.

What `PdfLayoutJob` does do is create a `PrintDocumentInfo.Builder` to set up a `PrintDocumentInfo` object indicating that:

- The output is a "document" (`CONTENT_TYPE_DOCUMENT`) versus a "photo" (`CONTENT_TYPE_PHOTO`) or "unknown" (`CONTENT_TYPE_UNKNOWN`). This information is passed to the `PrintService` that functions as a bridge between `PrintManager` and the printer, and the `PrintService` might optimize output based upon this setting (e.g., lower quality print output for a "document" instead of a "photo").
- The page count of the output is unknown (`PAGE_COUNT_UNKNOWN`). In principle, the page count is known, insofar as the PDF that will be printed is an asset baked into the app, and so we could hard-code the page count in addition to hard-coding other details (like the asset's filename).

The boolean second parameter to `onLayoutFinished()` is supposed to be `true` if the layout changed, `false` otherwise. In practice, the value does not seem to matter on the first `onLayout()` call. The implementation here compares the two `PrintAttributes` objects using `equals()`.

The last piece is the `PdfWriteJob`, which performs the work required of the `onWrite()` callback:

```
  private static class PdfWriteJob extends WriteJob {
    PdfWriteJob(PageRange[] pages, ParcelFileDescriptor destination,
                CancellationSignal cancellationSignal,
                WriteResultCallback callback, Context ctxt) {
      super(pages, destination, cancellationSignal, callback, ctxt);
    }

    @Override
    public void run() {
      InputStream in=null;
      OutputStream out=null;

      try {
        in=ctxt.getAssets().open("cover.pdf");
        out=new FileOutputStream(destination.getFileDescriptor());

        byte[] buf=new byte[16384];
        int size;

        while ((size=in.read(buf)) >= 0
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
          && !cancellationSignal.isCanceled()) {
        out.write(buf, 0, size);
      }

      if (cancellationSignal.isCanceled()) {
        callback.onWriteCancelled();
      }
      else {
        callback.onWriteFinished(new PageRange[] { PageRange.ALL_PAGES });
      }
    }
    catch (Exception e) {
      callback.onWriteFailed(e.getMessage());
      Log.e(getClass().getSimpleName(), "Exception printing PDF", e);
    }
    finally {
      try {
        in.close();
        out.close();
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(),
              "Exception cleaning up from printing PDF", e);
      }
    }
  }
}
```

At its core, `PdfWriteJob` simply writes our PDF (culled from a `cover.pdf` asset) to an `OutputStream`. The `OutputStream` is built from the `ParcelFileDescriptor`, indicating where the PDF content should be written to.

The `InputStream`-to-`OutputStream` "bucket brigade" is augmented with checks on the `CancellationSignal`, to abandon the loop if the print job was canceled by the user. At the end, we call one of three methods on the `WriteResultCallback`:

- `onWriteCancelled()` if the `CancellationSignal` indicates that the job was canceled
- `onWriteFinished()` if everything succeeded
- `onWriteFailed()` (with an error message) if there was some problem, such as failed I/O

`PdfWriteJob` has access to a `PageRange` array, representing the particular pages out of a larger document to be printed. The parameter to `onWriteFinished()` is another `PageRange` array that should indicate what pages were printed. Once again, since the PDF is fixed, `PdfWriteJob` ignores the input `PageRange` array, and it indicates that we wrote all pages (`PageRange.ALL_PAGES`) in the output. In principle, if you have more control over your environment, you should only print the requested pages, in which

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

case the output parameter to `onWriteFinished()` might be the same array as was passed into `onWrite()`.

Using PdfDocumentAdapter

Back in `MainActivity`, the "PDF" action bar overflow item triggers a call to `print()` on the `PrintManager`, supplying our `PdfDocumentAdapter` and another empty `PrintAttributes`:

```
        print("Test PDF",
              new PdfDocumentAdapter(getApplicationContext()),
              new PrintAttributes.Builder().build());
```

The `PdfDocumentAdapter` needs a `Context`, in order to access the `cover.pdf` asset. If your PDF file is being generated, or is saved as a file on external storage, you would not need this. Since it is theoretically possible that our activity could be destroyed while the printing is going on in background threads, rather that briefly leak an `Activity`, we provide the `Application Context` to `PdfDocumentAdapter`, as that is a singleton and cannot be leaked.

The result of all of this is that when the user chooses the "PDF" action bar overflow item, the book cover copy is printed.

Printing Using a Canvas

What Google *really* wants you to do — if bitmaps and HTML are insufficient – is to create PDF documents using `PdfPrintedDocument` and a `Canvas`.

The concept is simple:

- Create a `PrintedPdfDocument` instance, given a `PrintAttributes` that describes the page size, margins, etc.
- Call `startPage()` to add a page to the document, which returns a `PdfDocument.Page`
- Call `getCanvas()` on the `Page` and use the standard Android 2D drawing APIs to draw lines, text, shaded areas, and so forth
- Call `finishPage()` on the `PdfPrintedDocument` when you are done rendering that page
- Repeat the preceding three steps for all needed pages

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Call `writeTo()` on the `PrintedPdfDocument` to write the PDF to an `OutputStream`, such as the one you get from the `ParcelFileDescriptor` in the `onWrite()` callback of your `PrintDocumentAdapter`
- Call `close()` on the `PrintedPdfDocument` when you are done

For example, let's look at the `onWrite()` implementation used by `PrintHelper` to print a bitmap:

```
@Override
public void onWrite(PageRange[] pageRanges, ParcelFileDescriptor fileDescriptor,
                    CancellationSignal cancellationSignal,
                    WriteResultCallback writeResultCallback) {
    PrintedPdfDocument pdfDocument = new PrintedPdfDocument(mContext,
            mAttributes);
    try {

        Page page = pdfDocument.startPage(1);
        RectF content = new RectF(page.getInfo().getContentRect());

        // Compute and apply scale to fill the page.
        Matrix matrix = getMatrix(mBitmap.getWidth(), mBitmap.getHeight(),
                content, fittingMode);

        // Draw the bitmap.
        page.getCanvas().drawBitmap(mBitmap, matrix, null);

        // Finish the page.
        pdfDocument.finishPage(page);

        try {
            // Write the document.
            pdfDocument.writeTo(new FileOutputStream(
                    fileDescriptor.getFileDescriptor()));
            // Done.
            writeResultCallback.onWriteFinished(
                    new PageRange[]{PageRange.ALL_PAGES});
        } catch (IOException ioe) {
            // Failed.
            Log.e(LOG_TAG, "Error writing printed content", ioe);
            writeResultCallback.onWriteFailed(null);
        }
    } finally {
        if (pdfDocument != null) {
            pdfDocument.close();
        }
        if (fileDescriptor != null) {
            try {
                fileDescriptor.close();
            } catch (IOException ioe) {
                /* ignore */
            }
        }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
}
```

(note: the preceding code snippet is Copyright (C) 2013 The Android Open Source Project)

Here, they:

- Create the `PrintedPdfDocument`
- Add a page using `startPage()`
- Calculate a scaling `Matrix` based upon the image size, the page size, and the scale type (`FIT` or `FILL`)
- Draw the bitmap on the `Canvas` using that `Matrix`
- Finish the page
- Write the result to an `OutputStream` for the supplied `ParcelFileDescriptor`
- Close the document

Curiously, they do not do this work in a background thread, though the `onLayout()` implementation does use a background thread (since the image `Uri` may require an Internet download).

If you are comfortable with the `Canvas` API, writing PDF pages is much the same as drawing to your custom `View`. On the other hand, Android's `Canvas` API is not the same as any other drawing system's API, so there will be distinct differences from any other 2D drawing API that you might have used previously.

Print Jobs

The `print()` method that we have been calling on `PrintManager` returns a `PrintJob`, representing the print job. This object has a number of status inquiry methods, including (in rough order of when the events occur):

- `isStarted()`
- `isQueued()` (i.e., waiting for the print system to process it)
- `isBlocked()` (i.e., permanently stuck, but needs to be canceled)
- `isCompleted()`
- `isFailed()`
- `isCancelled()`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It also has a `cancel()` method that you can call to cancel the print job (e.g., based on user request). `PrintJob` also offers a `restart()` method that you can use to re-try a failed (but not canceled) print job.

What `PrintJob` does not have is a listener interface to be proactively notified when the job changes state.

`PrintManager` also has `getPrintJobs()`, which will return a list of the `PrintJob` objects representing the jobs you have requested in this process, rather than having to keep track of all of those yourself.

Printing, Threads, and Services

If you are going to create a report in HTML, you will want to consider doing that work in an `AsyncTask`'s `doInBackground()` method, so the I/O involved in creating the report happens in the background. However, `PrintManager` requires that `print()` be called on the main application thread, so you would call `print()` from `onPostExecute()` of the `AsyncTask`.

Similarly, if you are creating your own `PrintDocumentAdapter`, you will want to consider moving the `onLayout()` and `onWrite()` work into background threads, such as is illustrated in the sample app via `ThreadedPrintDocumentAdapter`.

The problem with bare threads or an `AsyncTask` is that they do not indicate to Android that your process is still doing some work. It is possible that the user could request that you print something, then switch to another app (e.g., HOME, recent-tasks list). Android might consider your process to be relatively low priority and could terminate it before your print job completes.

The obvious solution is to involve a service, perhaps even a foreground service, to indicate to Android that your process is doing work that the user will notice if it does not complete. You could start the service when you do the print job, and then stop the service when the print job is completed, to return your process to normal priority.

However, actually having a service do the printing is a serious pain:

- `WebView`'s `PrintDocumentAdapter` really wants the `Context` that created the `WebView` to be an `Activity`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- The key parameters to `onLayout()` and `onWrite()` are not `Parcelable` and so cannot be passed in `Intent` extras via `startService()` to the service

One possibility would be to create a `PrintJobMonitorService`, which is what the sample app does. `PrintJobMonitorService` takes advantage of that `listPrintJobs()` method on `PrintManager` to keep tabs on all of our requested print jobs. So long as there is one or more print jobs in an active state, the service keeps running. Otherwise, the service stops. Hence, while the service is not actually doing the printing, it is running while the printing is going on, flagging to the OS to leave our process alone during this critical juncture.

```
package com.commonsware.android.print;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.SystemClock;
import android.print.PrintJob;
import android.print.PrintJobInfo;
import android.print.PrintManager;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class PrintJobMonitorService extends Service implements Runnable {
  private static final int POLL_PERIOD=3;
  private PrintManager mgr=null;
  private ScheduledExecutorService executor=
      Executors.newSingleThreadScheduledExecutor();
  private long lastPrintJobTime=SystemClock.elapsedRealtime();

  @Override
  public void onCreate() {
    super.onCreate();

    mgr=(PrintManager)getSystemService(PRINT_SERVICE);
    executor.scheduleAtFixedRate(this, POLL_PERIOD, POLL_PERIOD,
                                 TimeUnit.SECONDS);
  }

  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    return(super.onStartCommand(intent, flags, startId));
  }

  @Override
  public void onDestroy() {
    executor.shutdown();

    super.onDestroy();
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void run() {
    for (PrintJob job : mgr.getPrintJobs()) {
      if (job.getInfo().getState() == PrintJobInfo.STATE_CREATED
          || job.isQueued() || job.isStarted()) {
        lastPrintJobTime=SystemClock.elapsedRealtime();
      }
    }

    long delta=SystemClock.elapsedRealtime() - lastPrintJobTime;

    if (delta > POLL_PERIOD * 2) {
      stopSelf();
    }
  }

  @Override
  public IBinder onBind(Intent intent) {
    return(null);
  }
}
```

`PrintJobMonitorService` uses a single-thread `ScheduledExecutorService`, to get control every three seconds in its `run()` method. The `run()` method iterates over the `PrintJob` objects associated with our app and looks for any that are in one of three states:

- "started", meaning that printing has begun
- "queued", meaning that the user has accepted the print dialog values, but printing has not yet started
- "created", meaning that the job has been created, but it is not yet considered queued, such as when the print dialog is up on the screen

The first two states have simple test methods on `PrintJob` (`isStarted()` and `isQueued()`). The "created" state does not, for some reason, so we have to get the underlying `PrintJobInfo` object and manually check its state (`getState()`) to see if it is started (`PrintJobInfo.STATE_STARTED`).

`PrintJobMonitorService` tracks the last time we saw an in-progress print job. If we have gone through two three-second polling periods without any in-progress print jobs, the service assumes that it is no longer needed and calls `stopSelf()`.

Printing Prior to Android 4.4

Before Android 4.4, printing in Android was limited and clunky.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The primary approach was to use Google Cloud Print. In effect, Google Cloud Print is a Web-managed print server. You would teach Google how to talk to your printers, and then any authorized device could print to those printers. By sharing your content (particularly PDFs) via `ACTION_SEND`, the user could choose Google Cloud Print as an option if they had Google Cloud Print set up for their device and printer. Note that the Android 4.4 printing framework includes a `PrintService` that works with Google Cloud Print, so users who have set up Google Cloud Print can still use it even with the new printing framework.

Various printer manufacturers or third parties also created their own apps that would fill a similar role, albeit perhaps working with printers on the local network. Or, you could write your own low-level code to talk to a network printer via relevant printing protocols like IPP, though this would be unpleasant at best.

HTML Generation

Earlier in this chapter, we saw how to print HTML. However, the HTML we printed was loaded from a URL. That is fine, but, as with printing bitmaps, it may not be a very popular scenario. What will be more likely is that you want to print some sort of report, generated on the device. And, since printing using the `Canvas` is a bit complicated, creating the report via HTML may be an easier route to take.

The typical approach for this involves creating an HTML template that sets up the basic page (e.g., references to CSS), then uses some sort of "macros" in the template to indicate portions that should be replaced dynamically with something from outside of the template.

This approach has been used since the early days of the original "dot-com revolution" of the 1990's, pioneered by tools like Cold Fusion. In Java, there are any number of available template engines.

However, for HTML, it is reasonably likely that a Web designer is going to want to get involved, to style the report. Ideally, you choose a template engine that is either something the designer is already using, or is one that is something the designer might wish to use elsewhere in the future. Forcing the designer to learn some new template syntax, just for the purposes of creating these reports, may not be the best use of the designer's time (or your time, for answering all of the designer's questions).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

One of the more popular template structures used today use braces (a.k.a., curly brackets) as the macro delimiters (e.g., `{{ something }}`). In particular, the macro syntax popularized by mustache is used by many template engine implementations. There is a very good chance that your Web designer already has used mustache-style templates, or at least has heard about them.

And, conveniently enough, there is a Java implementation – jmustache — that is Android-friendly. The sample app in this chapter implements a "TPS Report" that is generated from a mustache template using jmustache.

Adding jmustache To Your App

The "Get It" section of the jmustache documentation contains up-to-date instructions for adding it to a project.

Developers using Gradle for Android — including Android Studio users – should reference the Maven Central artifact (`com.samskivert:jmustache`) from `build.gradle`.

Developers using Eclipse or Ant can download the JAR file and add it to their project's `libs/` directory.

Writing the Report Template

A report template for jmustache can be a `String` or a `Reader`, with the latter allowing you to pull in files, assets, or raw resources (the latter two via an `InputStreamReader`).

In the case of the sample app, the template is small, and is packaged as a string resource. However, since the template involves HTML tags, we have to use `CDATA` notation to allow those tags to be left alone within the XML of the string resource:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <string name="report_body">
<![CDATA[
<html>
<body>
<h1>TPS Report for: {{reportDate}}</h1>
<p>Here are the contents of this week\'s TPS report:</p>
<p>{{message}}</p>
<p>If you have any questions regarding this report, please
do <b>not</b> ask Mark Murphy.</p>
</body>
</html>
]]>
```

Figure 686: TPS Report Mustache Template

The template contains `{{reportDate}}` and `{{message}}` variables to be replaced at runtime with dynamic data from our app. Also note that, despite the `CDATA`, we still need to escape the apostrophe with a leading backslash (`\'`).

Creating a Report Context

What will fill in the `{{reportDate}}` and `{{message}}` variables will be values from a "context". Here, "context" is not referring to `Context`, but rather an object that we pass to jmustache to serve as the source of data to blend into the report.

jmustache has fairly flexible rules for how it can resolve template variables, including calling Java getter methods based on the variable names. Hence, we can create a "context" that has `getReportDate()` and `getMessage()` methods, such as the `TpsReportContext` class in the sample app:

```
private static class TpsReportContext {
  private static final SimpleDateFormat fmt=
      new SimpleDateFormat("yyyy-MM-dd", Locale.US);
  String msg;

  TpsReportContext(String msg) {
    this.msg=msg;
  }

  @SuppressWarnings("unused")
  String getReportDate() {
    return(fmt.format(new Date()));
  }

  @SuppressWarnings("unused")
  String getMessage() {
    return(msg);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }
```

Printing the Report

The "TPS Report" action bar overflow item eventually routes to a `printReport()` method on `MainActivity`:

```
  private void printReport() {
    Template tmpl=
        Mustache.compiler().compile(getString(R.string.report_body));
    WebView print=prepPrintWebView(getString(R.string.tps_report));

    print.loadData(tmpl.execute(new TpsReportContext(prose.getText()
                                                     .toString())),
                   "text/html", "UTF-8");
  }
```

The first statement creates a jmustache `Template` object representing the report template. This is created by getting the singleton `compiler()` from `Mustache`, and calling `compile()` on it to interpret the string resource. Note that since this `Template` only depends upon the string resource, we could cache the `Template`, rebuilding it only on configuration changes, if desired.

Note that we load the template on the main application thread, as `printReport()` is called from `onOptionsItemSelected()`. For a small string resource, that is OK. If you are loading a more complex report template, you will want to do that in a background thread.

The second statement mirrors one from printing the Web page from before, where we call `prepPrintWebView()` to lazy-create our `WebView` and set it up to print when the page is loaded. Here, we use a different print job name than before, one reflecting the fact that this is a TPS report.

Finally, we use `execute()` on the `Template` to generate our HTML for printing, then pass that HTML to the `loadData()` method on `WebView`. `execute()` takes our "context" `Object`, which in this case is an instance of our `TpsReportContext` class, with the value typed into the `EditText` widget in our UI as the "message" to go into the report.

Note that we `execute()` the `Template` on the main application thread as well as having loaded it on that thread in the first place. Once again, the more complex the report, the more likely it is that you will want to move this logic into a background

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

thread. However, remember that `print()` needs to be called on the main application thread.

The result is that the user gets a printed TPS report, containing today's date and whatever message they typed into the `EditText`.

PDF Generation Options

Perhaps you feel that generating HTML does not give you enough control, yet using the `Canvas` options directly was *too* much control. Perhaps you then think that generating a PDF to print, using something other than `PdfDocument`, is the right answer. Or perhaps you are generating a PDF for other reasons, such as to use with `ACTION_SEND` as output from your app.

You have two basic options for getting this PDF: generate it on the device, or offload the generation to a server.

There are various open source and commercial libraries for generating PDF on Android. The best-known open source Java PDF library – iText — has as dedicated Android version (iTextG), though the AGPL license may make it unsuitable for your use case. The commercial libraries range from fixed-price to per-device licenses. How much advantage these have over using `PrintedPdfDocument` from the Android SDK depends upon your needs.

If the bulk of the data needed for generating the PDF resides on a server, rather than downloading that data and using an underpowered Android device to create the PDF, you could upload the device-specific data to the server, have it create the PDF, and download the result from the server. There are plenty of server-side PDF generation tools, ranging from open source (e.g., wkhtmltopdf, unoconv, prawn) to commercial (e.g., Prince, used to generate the PDF edition of this book). You also get to work in your preferred programming language, in case that is not Java, and perhaps leverage the PDF generation logic for other uses (e.g., generate reports from your Web app).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Other System Settings and Services

Android offers a number of system services, usually obtained by `getSystemService()` from your `Activity`, `Service`, or other `Context`. These are your gateway to all sorts of capabilities, from settings to volume to WiFi. Throughout the course of this book, we have seen several of these system services. In this chapter, we will take a look at others that may be of value to you in building compelling Android applications.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Setting Expectations

If you have an Android device, you probably have spent some time in the Settings application, tweaking your device to work how you want – ringtones, WiFi settings, USB debugging, etc. Many of those settings are also available via `Settings` class (in the `android.provider` package), and particularly the `Settings.System` and `Settings.Secure` public inner classes.

Basic Settings

`Settings.System` allows you to get and, with the `WRITE_SETTINGS` permission, alter these settings. As one might expect, there are a series of typed getter and setter methods on `Settings.System`, each taking a key as a parameter. The keys are class constants, such as:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. `INSTALL_NON_MARKET_APPS` to control whether you can install applications on a device from outside of the Play Store
2. `HAPTIC_MODE_ENABLED` to control whether the user receives "haptic feedback" (vibrations) from things like the MENU button
3. `ACCELEROMETER_ROTATION` to control whether the screen orientation will change based on the position of the device

The SystemServices/Settings sample project has a `SettingsSetter` sample application that displays a checklist:

```
<?xml version="1.0" encoding="utf-8"?>
<ListView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/list"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
/>
```

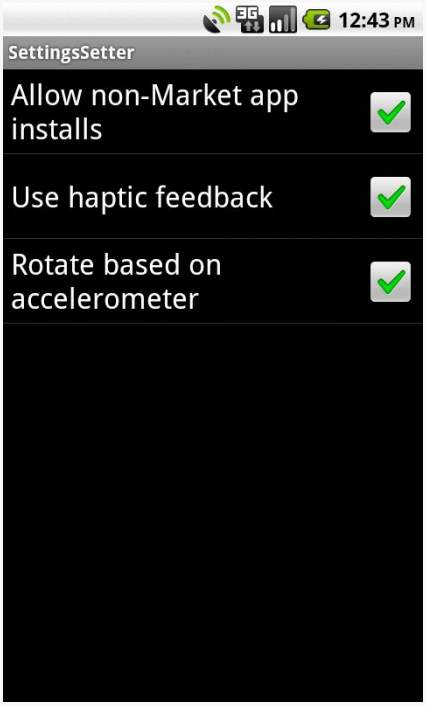


Figure 687: The SettingsSetter application

The checklist itself is filled with a few `BooleanSetting` objects, which map a display name with a `Settings.System` key:

```
  static class BooleanSetting {
    String key;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  String displayName;
  boolean isSecure=false;

  BooleanSetting(String key, String displayName) {
    this(key, displayName, false);
  }

  BooleanSetting(String key, String displayName,
                 boolean isSecure) {
    this.key=key;
    this.displayName=displayName;
    this.isSecure=isSecure;
  }

  @Override
  public String toString() {
    return(displayName);
  }

  boolean isChecked(ContentResolver cr) {
    try {
      int value=0;

      if (isSecure) {
        value=Settings.Secure.getInt(cr, key);
      }
      else {
        value=Settings.System.getInt(cr, key);
      }

      return(value!=0);
    }
    catch (Settings.SettingNotFoundException e) {
      Log.e("SettingsSetter", e.getMessage());
    }

    return(false);
  }

  void setChecked(ContentResolver cr, boolean value) {
    try {
      if (isSecure) {
        Settings.Secure.putInt(cr, key, (value ? 1 : 0));
      }
      else {
        Settings.System.putInt(cr, key, (value ? 1 : 0));
      }
    }
    catch (Throwable t) {
      Log.e("SettingsSetter", "Exception in setChecked()", t);
    }
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Three such settings are put in the list:

```
    settings.add(new BooleanSetting(Settings.System.INSTALL_NON_MARKET_APPS,
                                    "Allow non-Market app installs",
                                    true));
    settings.add(new BooleanSetting(Settings.System.HAPTIC_FEEDBACK_ENABLED,
                                    "Use haptic feedback",
                                    false));
    settings.add(new BooleanSetting(Settings.System.ACCELEROMETER_ROTATION,
                                    "Rotate based on accelerometer",
                                    false));
```

As the checkboxes are checked and unchecked, the values are passed along to the settings themselves:

```
  @Override
  protected void onListItemClick(ListView l, View v,
                                 int position, long id) {
    super.onListItemClick(l, v, position, id);

    BooleanSetting s=settings.get(position);

    s.setChecked(getContentResolver(),
                 l.isItemChecked(position));
  }
```

The `SettingsSetter` activity also has an option menu containing four items:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/app"
    android:title="Application"
    android:icon="@android:drawable/ic_menu_manage" />
  <item android:id="@+id/security"
    android:title="Security"
    android:icon="@android:drawable/ic_menu_close_clear_cancel" />
  <item android:id="@+id/wireless"
    android:title="Wireless"
    android:icon="@android:drawable/ic_menu_set_as" />
  <item android:id="@+id/all"
    android:title="All Settings"
    android:icon="@android:drawable/ic_menu_preferences" />
</menu>
```

These items correspond to four activity `Intent` values identified by the `Settings` class:

```
    menuActivities.put(R.id.app,
                       Settings.ACTION_APPLICATION_SETTINGS);
    menuActivities.put(R.id.security,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                      Settings.ACTION_SECURITY_SETTINGS);
  menuActivities.put(R.id.wireless,
                      Settings.ACTION_WIRELESS_SETTINGS);
  menuActivities.put(R.id.all,
                      Settings.ACTION_SETTINGS);
```

When an option menu is chosen, the corresponding activity is launched:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  String activity=menuActivities.get(item.getItemId());

  if (activity!=null) {
    startActivity(new Intent(activity));

    return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

This way, you have your choice of either directly manipulating the settings or merely making it easier for users to get to the Android-supplied activity for manipulating those settings.

Secure Settings

You will notice that if you use the above code and try changing the Play Store setting, it does not seem to take effect. And, if you look at the LogCat output, you will see complaints.

Once upon a time, you could modify this setting, and others like it.

Now, though, these settings are ones that Android deems "secure". The constants have been moved from `Settings.System` to `Settings.Secure`, though the old constants are still there, flagged as deprecated.

These so-called "secure" settings are ones that Android does not allow applications to change. While theoretically the `WRITE_SECURE_SETTINGS` permission resolves this problem, ordinary SDK applications cannot hold that permission. The only option is to display the official Settings activity and let the user change the setting.

API Level 17 takes things one step further, moving a number of settings out of `Settings.System` and `Settings.Secure` and placing them in a new `Settings.Global`. Like `Settings.Secure`, ordinary SDK apps cannot modify these

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

settings, as they are secured by `WRITE_SECURE_SETTINGS`. The distinction between `Settings.Secure` and `Settings.Global` comes with respect to scope: `Settings.Secure` is on a per-user basis (for devices set up with multiple users), and `Settings.Global` is device-wide.

Can You Hear Me Now? OK, How About Now?

The fancier the device, the more complicated controlling sound volume becomes.

On a simple MP3 player, there is usually only one volume control. That is because there is only one source of sound: the music itself, played through speakers or headphones.

In Android, though, there are several sources of sounds:

1. Ringing, to signify an incoming call
2. Voice calls
3. Alarms, such as those raised by the Alarm Clock application
4. System sounds (error beeps, USB connection signal, etc.)
5. Music, as might come from the MP3 player

Android allows the user to configure each of these volume levels separately. Usually, the user does this via the volume rocker buttons on the device, in the context of whatever sound is being played (e.g., when on a call, the volume buttons change the voice call volume). Also, there is a screen in the Android Settings application that allows you to configure various volume levels.

The `AudioService` in Android allows you, the developer, to also control these volume levels, for all five "streams" (i.e., sources of sound). In the `SystemServices/Volume` sample project, we create a `Volumizer` application that displays and modifies all five volume levels.

Attaching SeekBars to Volume Streams

The standard widget for allowing choice along a range of integer values is the `SeekBar`, a close cousin of the `ProgressBar`. `SeekBar` has a thumb that the user can slide to choose a value between `0` and some maximum that you set. So, we will use a set of five `SeekBar` widgets to control our five volume levels.

First, we need to create a layout with a `SeekBar` per stream:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:stretchColumns="1">

  <TableRow
    android:paddingBottom="20px"
    android:paddingTop="10px">

    <TextView android:text="Alarm:"/>

    <SeekBar
      android:id="@+id/alarm"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow android:paddingBottom="20px">

    <TextView android:text="Music:"/>

    <SeekBar
      android:id="@+id/music"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow android:paddingBottom="20px">

    <TextView android:text="Ring:"/>

    <SeekBar
      android:id="@+id/ring"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow android:paddingBottom="20px">

    <TextView android:text="System:"/>

    <SeekBar
      android:id="@+id/system"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"/>
  </TableRow>

  <TableRow>

    <TextView android:text="Voice:"/>

    <SeekBar
      android:id="@+id/voice"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      android:layout_width="match_parent"
      android:layout_height="wrap_content"/>
  </TableRow>

</TableLayout>
```

Then, we need to wire up each of those bars in the `onCreate()` for `Volumizer`, calling an `initBar()` method for each of the five bars:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    mgr=(AudioManager)getSystemService(Context.AUDIO_SERVICE);

    alarm=(SeekBar)findViewById(R.id.alarm);
    music=(SeekBar)findViewById(R.id.music);
    ring=(SeekBar)findViewById(R.id.ring);
    system=(SeekBar)findViewById(R.id.system);
    voice=(SeekBar)findViewById(R.id.voice);

    initBar(alarm, AudioManager.STREAM_ALARM);
    initBar(music, AudioManager.STREAM_MUSIC);
    initBar(ring, AudioManager.STREAM_RING);
    initBar(system, AudioManager.STREAM_SYSTEM);
    initBar(voice, AudioManager.STREAM_VOICE_CALL);
  }
```

In `initBar()`, we set the appropriate size for the `SeekBar` bar via `setMax()`, set the initial value via `setProgress()`, and hook up an `OnSeekBarChangeListener` to find out when the user slides the bar, so we can set the volume on the stream via the `VolumeManager`.

The net result is that when the user slides a `SeekBar`, it adjusts the stream to match:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

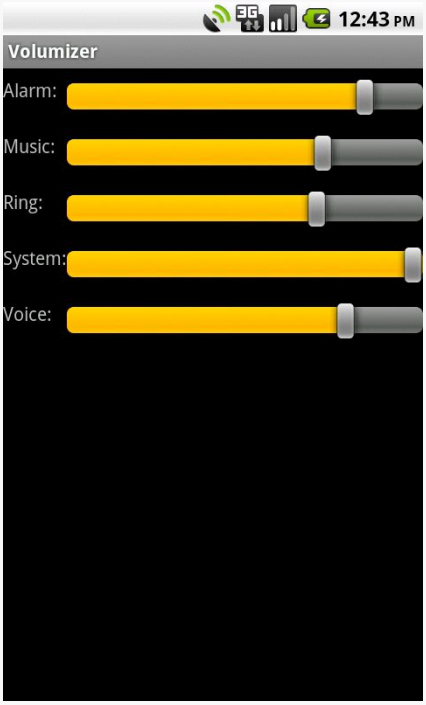


Figure 688: The Volumizer application

The Rest of the Gang

There are quite a few system services you can get from `getSystemService()`. Beyond the ones profiled in this chapter, you have access to:

1. `AccessibilityManager`, for being notified of key system events (e.g., activities starting) that might be relayed to users via haptic feedback, audio prompts, or other non-visual cues
2. `AccountManager`, for working with Android's system of user accounts and synchronization
3. `ActivityManager`, for getting more information about what processes and components are presently running on the device
4. `AlarmManager`, for scheduled tasks (a.k.a., "cron jobs"), covered elsewhere in this book
5. `ConnectivityManager`, for a high-level look as to what sort of network the device is connected to for data (e.g., WiFi, 3G)
6. `DevicePolicyManager`, for accessing device administration capabilities, such as wiping the device

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

7. `DownloadManager`, for downloading large files on behalf of the user, covered in the chapter on `Intents`
8. `DropBoxManager`, for maintaining your own ring buffers of logging information akin to LogCat
9. `InputMethodManager`, for working with input method editors
10. `KeyguardManager`, for locking and unlocking the keyguard, where possible
11. `LayoutInflater`, for inflating layout XML files into Views, covered elsewhere in this book
12. `LocationManager`, for determining the device's location (e.g., GPS), covered in the chapter on location tracking
13. `NotificationManager`, for putting icons in the status bar and otherwise alerting users to things that have occurred asynchronously, covered in the chapter on `Notification`
14. `PowerManager`, for obtaining `WakeLock` objects and such, covered elsewhere in this book
15. `SearchManager`, for interacting with the global search system – search in general is covered elsewhere in this book
16. `SensorManager`, for accessing data about sensors, such as the accelerometer
17. `TelephonyManager`, for finding out about the state of the phone and related data (e.g., SIM card details)
18. `UiModeManager`, for dealing with different "UI modes", such as being docked in a car or desk dock
19. `Vibrator`, for shaking the phone (e.g., haptic feedback)
20. `WifiManager`, for getting more details about the active or available WiFi networks
21. `WindowManager`, mostly for accessing details about the default display for the device

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Dealing with Different Hardware

While a lot of focus is placed on screen sizes, there are many other possible hardware differences among different Android devices. For example, some have telephony features, while others do not.

There is a three-phase plan for dealing with these variations:

1. **Filter** out devices that cannot possibly run your app successfully, so your app will not appear to them in the Play Store and they will be unable to install your app if obtained by other means
2. **React** to varying hardware that you can support, but perhaps might support differently (e.g., choosing a particular flash mode for a device having a camera with a flash)
3. **Cope** with device bugs or regressions that impact your application

This chapter will go through each of these topics.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Filtering Out Devices

Elsewhere in the book, we discussed a few manifest entries that will serve to filter out devices that cannot run your app:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- android:minSdkVersion in the <uses-sdk> element, to stipulate that devices must run a certain version of Android (or higher)
- <supports-screens> and <compatible-screens>, which indicate which screens sizes and densities you are capable of supporting

This section outlines other "advertisements" that you can put in the manifest to restrict which devices run your app.

uses-feature

The <uses-feature> element restricts your app to devices that have certain hardware features. For each element, you supply the name of a feature (e.g., android.hardware.telephony) and whether or not it is required:

```
<uses-feature
  android:name="android.hardware.camera"
  android:required="false" />
```

By default, android:required is set to true, so typically you will only see it in a manifest when it is set to false.

You might wonder why we would bother ever setting android:required to false. After all, that should have the same effect as not listing it at all. In practice, though, it has two major uses.

First, markets like the Play Store might highlight the fact that you *can* use a particular hardware capability, even though you do not strictly require it.

More importantly, you can use android:required="false" to *undo* a *requirement* that Android infers from your permissions. Requesting some permissions causes Android to assume — for backwards-compatibility reasons — that your app needs the affiliated hardware. For example, requesting the CAMERA permission causes Android to assume that you need a camera (android.hardware.camera) *and* that the camera support auto-focus (android.hardware.camera.autofocus). If, however, you are requesting the permission because you would like to use the hardware if available, but can live without it, you need to expressly add a <uses-feature> element declaring that the hardware feature is not required.

For example, in February 2010, the Motorola XOOM tablet was released. This was the first Android device that had the Play Store on it and truly had no telephony capability. As such, the XOOM would be filtered out of the then-Android Market (now Play Store) for any app that required permissions like SEND_SMS. Many

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

developers requested this permission, even though their apps could survive without SMS-sending capability. However, their apps were still filtered out if they did not have the <uses-feature> element declaring that telephony was not required.

You can find a table listing Android permissions and assumed hardware feature requirements in the Android developer documentation.

uses-configuration

The <uses-configuration> element is very reminiscent of <uses-feature>: it dictates hardware requirements. The difference is two-fold:

1. It focuses on hardware elements that represent different device configurations, meaning that you might use different resources for them
2. It allows you to specify *combinations* of capabilities that you need

There are three capabilities that you can require via <uses-configuration>:

1. The existence of a five-way navigation control, whether a specific type (D-pad, trackball, etc.) or any such control
2. The existence of a physical keyboard, whether a specific type (QWERTY, 12-key numeric keypad, etc.) or any such keyboard
3. A touchscreen

You can have as many <uses-configuration> elements as you need – any device that matches at least one such configuration will be eligible to install your app.

For example, the following <uses-configuration> element restricts your app to devices that have some sort of navigation control but do not necessarily have a touchscreen, such as a Android TV device:

```
<uses-configuration
  android:reqFiveWayNav="true"
  android:reqTouchScreen="notouch" />
```

uses-library

The <uses-library> element tells Android that your application wishes to use a particular *firmware*-supplied library. The most common case for this was Maps V1, which is shipped in the form of an SDK add-on and firmware library. This, however, has been deprecated for quite some time.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, there are other firmware libraries that you might need. These will typically be manufacturer-specific libraries, allowing your application to take advantage of particular beyond-the-Android-SDK capabilities of a particular device. For example, several Motorola Mobility devices ship with an "Enterprise Device Management" firmware library, as an extension of Android's own device admin APIs.

The Google Play Store will filter out your application from devices that lack a firmware library that you require via `<uses-library>`. If the user tries installing your app by some other means (e.g., download from a Web site), your app will fail to install on devices that lack the firmware library.

If you conditionally want the firmware library — you will use it if available but can cope if it is not — you can add `android:required="false"` to your `<uses-library>` element. That will allow your app to install and run on devices missing the library in question. Detecting whether or not the library exists in your process at runtime is a matter if using `Class.forName()` to see if you have access to some class from that library, where a `ClassNotFoundException` means that you do not have the library.

Runtime Capability Detection

Reacting to device capabilities is the second phase of dealing with different devices. Some features you might want (e.g., telephony for sending SMSes) but can live without. Other features may have subtle variations that you cannot filter against and therefore need to adapt to at runtime (e.g., possible picture resolutions off of a camera).

This section will cover various techniques for determining what a device can do, at runtime, so you can react accordingly.

Features

Any feature you do not make required via `<uses-feature>` can be detected at runtime by calling `hasSystemFeature()` on `PackageManager`. For example, if you would like to send SMS messages, but only on telephony-capable devices, you could have the following `<uses-feature>` element:

```
<uses-feature
  android:name="android.hardware.telephony"
  android:required="false" />
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Then, at runtime, you can call `hasSystemFeature("android.hardware.telephony")` on aPackageManager` instance to find out if, indeed, the device has telephony capability and sending SMSes should work.

Other Capabilities

Various subsystems have their own means of helping you determine what is possible or not:

- The `Camera` class, via `Camera.Parameters`, can let you know the capabilities of a camera (e.g., whether or not it has a flash, and what specific flash modes are supported).
- The `LocationManager` will help you determine what location providers are available that meet your `Criteria`.
- The sensor subsystem lets you find out what sensors are installed, either overall or for a particular type (e.g., accelerometer).

Dealing with Device Bugs

Alas, devices are not perfect. Even though the Compatibility Test Suite attempts to ensure that all Android devices legitimately running the Play Store faithfully implement the Android SDK, some device manufacturers make changes that introduce bugs.

Just as Web developers can "sniff" on the `User-Agent` HTTP header to determine what sort of browser is requesting a page, you can use the `Build` class to determine what sort of device is running your app. If you encounter problems with a specific device, you may be able to use `Build` to identify that device at runtime and "route around the damage".

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Integration and Introspection

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Responding to URLs

You may have noticed that Android supports a `market:` URL scheme. Web pages can use such URLs so that, if they are viewed on an Android device's browser, the user can be transported to a Play Store page, perhaps for a specific app or a list of apps for a publisher.

Fortunately, that mechanism is not limited to Android's code — you can get control for various other types of links as well. You do this by adding certain entries to an activity's `<intent-filter>` for an `ACTION_VIEW` `Intent`.

However, be forewarned that this capability is browser-specific. What works on the original Android "Browser" app and Google's Chrome may not necessarily work on Firefox for Android or other browsers.

Prerequisites

Understanding this chapter requires that you have read the chapter on `Intent` filters.

Manifest Modifications

First, any `<intent-filter>` designed to respond to browser links will need to have a `<category>` element with a name of `android.intent.category.BROWSABLE`. Just as the `LAUNCHER` category indicates an activity that should get an icon in the launcher, the `BROWSABLE` category indicates an activity that wishes to respond to browser links.

You will then need to further refine which links you wish to respond to, via a `<data>` element. This lets you describe the URL and/or MIME type that you wish to respond

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

to. For example, here is the `AndroidManifest.xml` file from the Introspection/URLHandler sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.urlhandler"
  android:versionCode="1"
  android:versionName="1.0">

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"/>

  <application
    android:icon="@drawable/cw"
    android:label="@string/app_name">
    <activity
      android:name="URLHandler"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.VIEW"/>

        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>

        <data android:mimeType="application/pdf"/>
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.VIEW"/>

        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>

        <data
          android:host="www.this-so-does-not-exist.com"
          android:path="/something"
          android:scheme="http"/>
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.android.MY_ACTION"/>

        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
      </intent-filter>
    </activity>
  </application>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
</manifest>
```

Here, we have four `<intent-filter>` elements for our one activity:

- The first is a standard "put an icon for me in the launcher, please" filter, with the `LAUNCHABLE` category
- The second claims that we handle PDF files (MIME type of `application/pdf`), and that we will respond to browser links (`BROWSABLE` category)
- The third claims that we will handle any HTTP request (scheme of `"http"`) for a certain Web site (host of `"www.this-so-does-not-exist.com"` and path of `/something`), and that we will respond to browser links (`BROWSABLE` category)
- The last is a custom action, for which we will generate a URL that Android will honor, and that we will respond to browser links (`BROWSABLE` category) — we will examine this more closely in the next section

What happens for the first two links varies based on browser.

The original Android "Browser" app, and Google Chrome, will do the following:

- Tapping the link to the PDF, on Android 2.3+, will trigger a download of the PDF. When the user taps on the downloaded file (e.g., from the `Notification` in the status bar), the user will have `URLHandler` as one of the options in the chooser to view the PDF file.
- Tapping the link to `http://www.this-so-does-not-exist.com/something` will bring up a chooser showing all available Web browser, plus `URLHandler`, as expected

Firefox for Android will treat the PDF link the same way. However, Firefox for Android does not check the URL for the second link to see if there is anything else supporting `ACTION_VIEW` for the URL, and so it always loads up the Web page. You see this effect with the link to Barcode Scanner as well — even though a device has Barcode Scanner installed, Firefox never offers that as an option.

Creating a Custom URL

Responding to MIME types makes complete sense... if we implement something designed to handle such a MIME type.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Responding to certain schemes, hosts, paths, or file extensions is certainly usable, but other than perhaps the file extension approach, it makes your application a bit fragile. If the site changes domain names (even a sub-domain) or reorganizes its site with different URL structures, your code will break.

If the goal is simply for you to be able to trigger your own application from your own Web pages, though, the safest approach is to use an `intent:` URL. These can be generated from an `Intent` object by calling `toUri(Intent.URI_INTENT_SCHEME)` on a properly-configured `Intent`, then calling `toString()` on the resulting `Uri`.

For example, the `intent:` URL for the fourth `<intent-filter>` from above is:

```
intent:#Intent;action=com.commonsware.android.MY_ACTION;end
```

This is not an official URL scheme, any more than `market:` is, but it works for Android devices. When the Android built-in Browser encounters this URL, it will create an `Intent` out of the URL-serialized form and call `startActivity()` on it, thereby starting your activity. Chrome also supports this URL structure. Firefox for Android does not, indicating instead that it cannot recognize the URL.

Reacting to the Link

Your activity can then examine the `Intent` that launched it to determine what to do. In particular, you will probably be interested in the `Uri` corresponding to the link — this is available via the `getData()` method. For example, here is the `URLHandler` activity for this sample project:

```
package com.commonsware.android.urlhandler;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.TextView;

public class URLHandler extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    TextView uri=(TextView)findViewById(R.id.uri);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    if (Intent.ACTION_MAIN.equals(getIntent().getAction())) {
      String intentUri=(new Intent("com.commonsware.android.MY_ACTION"))
                          .toUri(Intent.URI_INTENT_SCHEME)
                          .toString();

      uri.setText(intentUri);
      Log.w("URLHandler", intentUri);
    }
    else {
      Uri data=getIntent().getData();

      if (data==null) {
        uri.setText("Got com.commonsware.android.MY_ACTION Intent");
      }
      else {
        uri.setText(getIntent().getData().toString());
      }
    }
  }

  public void visitSample(View v) {
    startActivity(new Intent(Intent.ACTION_VIEW,
                             Uri.parse("http://commonsware.com/sample")));
  }
}
```

This activity's layout has a `TextView` (uri) for showing a `Uri` and a `Button` to launch a page of links, found on the CommonsWare site (`http://commonsware.com/sample`). The `Button` is wired to call `visitSample()`, which just calls `startActivity()` on the aforementioned URL to display it in the Browser.

When the activity starts up, though, it first loads up the `TextView`. What goes in there depends on how the activity was launched:

1. If it was launched via the launcher (e.g., the action is `MAIN`), then we display in the `TextView` the `intent:` URL shown in the previous section, generated from an `Intent` object designed to trigger our fourth `<intent-filter>`. This also gets dumped to LogCat, and is how the author got this URL in the first place to put on the sample Web page of links.
2. If it was not launched via the launcher, it was launched from a Web link. If the `Uri` from the launching `Intent` is `null`, though, that means the activity was launched via the custom `intent:` URL (which only has an action string), so we put a message in the `TextView` to match.
3. Otherwise, the `Uri` from the launching `Intent` will have something we can use to process the link request. For the PDF file, it will be the local path to the downloaded PDF, so we can open it. For the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`www.this-so-does-not-exist.com` URL, it will be the URL itself, so we can process it our own way.

Note that for the PDF case, clicking the PDF link in the Browser will download the file in the background, with a `Notification` indicating when it is complete. Tapping on the entry in the notification drawer will then trigger the `URLHandler` activity.

Also, bear in mind that the device may have multiple handlers for some URLs. For example, a device with a real PDF viewer will give the user a choice of whether to launch the downloaded PDF in the real view or `URLHandler`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Plugin Patterns

Plugins have historically been a popular model for extending the functionality of a base application. Browsers, for example, have long used plugins for everything from playing Flash animations to displaying calendars.

While Android does not have a specific "plugin framework", many techniques exist in Android to create plugins. Which of these patterns is appropriate for you will depend upon the nature of the host application and, more importantly, on the nature of the plugin. This chapter will explore some of these plugin patterns.

Prerequisites

Having read the chapters on app widgets (to be exposed to `RemoteViews`) and the `Loader` framework would be useful, though neither is essential for grasping the core concepts presented in this chapter. Similarly, this chapter has a case study that covers a lockscreen widget, so knowing a bit about those will help, but is not absolutely essential. Another sample involves the use of custom permissions, which are subject to a vulnerability covered in another chapter.

Definitions, Scenarios, and Scope

For the purposes of this chapter, a "plugin model" refers to an app (the plugin "host") that is being extended by other apps (the "plugins") that are largely dedicated to that job.

Certainly, there are plenty of ways that apps can work together without one being a plugin to another. The user's Web browser is not a plugin of your app when you call `startActivity()` to view a Web page, for example.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

By contrast, the Locale app can be extended via plugins, written either by two forty four a.m. (the authors of Locale) or third parties. These plugins have no real value to the user other than by how they improve what Locale itself can do. This sort of structure, therefore, qualifies as a plugin model.

In particular, this chapter will focus on two general scenarios for wanting a plugin model, though others certainly exist:

1. You want to allow third parties to extend the capability of your app, much as two forty four a.m. wanted with Locale, or
2. You want to reduce the number of permissions in your core app by delegating some permissions to plugins, so users can "opt into" those permissions

The Keys to Any Plugin System

There are four essential ingredients for any plugin model:

1. Somehow, the user has to be able to find, download, and install plugins for the host.
2. Somehow, the host app has to know what plugins are installed and available for use.
3. Somehow, the host app and the plugin need to communicate, usually through one form or another of inter-process communication (IPC)
4. All of this needs to be done without compromising the user's privacy or security

Depending upon the nature of the host app and plugin system, there may need to be additional ingredients (e.g., allowing users to configure the behavior of plugins).

Discovery… By the User

A popular thought experiment is:

> If a tree falls in a forest and no one is around to hear it, does it make a sound?

The analog to plugins is:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

> If an app offers a plugin model, and the user cannot find any plugins, is there really a plugin model?

Somehow, users need to know about available plugins, and frequently that means that *you* will need to help steer them towards those plugins.

If you are focused solely on distributing through the Play Store, you could invite all of your plugin authors to use some particular keyword or phrase, likely to be unique for your plugins, then use `market://search?q=...&c=apps` (with `...` replaced by your keyword or phrase) as a `Uri` for an `ACTION_VIEW Intent` passed to `startActivity()`. This will show the user a list of all apps on the Play Store with that keyword or phrase. For example, SONY suggested that developers writing extensions for the SONY SmartWatch use "smartwatch" as a keyword.

Of course, you are welcome to maintain your own roster of available plugins, where your app can download that roster as needed and display the candidates to your users. For example, you might have a JSON file on your Web server at a well-known URL that contains the current lineup of available plugins.

Or, you are welcome to simply offer this sort of information via your Web site, not from within your app. Depending upon how frequently users will be visiting your Web site, this may or may not be helpful to them, but it may be simpler than doing something custom built into your app. For example, you could maintain a simple static Web page with links to the plugins.

Discovery… By Your App

Once a user installs one or more plugins, your plugin host app needs to know that they are there. Continuing with the thought experiments:

> If an app offers a plugin model, but fails to recognize any plugins, is there really a plugin model?

Conversely, once a user *removes* a plugin, your host app needs to know about that as well, so that you do not try to use a plugin that no longer exists.

There are any number of possible strategies for finding available plugins; the following sections outline a few candidates.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Broadcast-and-Response

One approach is to send a custom broadcast `Intent`, at relevant points in time, that is an advertisement to plugins, saying "Hey! Tell me that you exist!". Plugins, as part of your instructions for writing a plugin, are obligated to respond to that broadcast by doing something to let you know about them, such as:

- Sending their own broadcast back to your host app, providing details about the plugin
- Inserting or updating an entry in a host-published `ContentProvider`
- Sending a command to a host-supplied `IntentService`
- Etc.

Any previously-existing plugins that do not respond within some specific period of time are considered "gone", possibly with the host app using `PackageManager` and `getPackageInfo()` to confirm that it is gone.

This is fairly easy to set up, but suffers from non-deterministic timing of broadcasts. The host app can only guess when the broadcast has had enough time to reach all of the plugins and gather responses. It also forces all of those plugin apps to run (to respond to the broadcast), which will cause Android to eject other apps from memory, possibly irritating the user.

Another limitation is that a newly-installed plugin will not respond to a broadcast, on Android 3.1+, until something manually runs one of that plugin's components, such as the user tapping on the plugin's activity in the launcher. Not only does this require the plugin to have such an activity (which might not otherwise be needed), but it means that the plugin is useless until this happens. We will discuss this issue a bit more later in this chapter.

Scanning with PackageManager

You could skip the broadcast and directly use `PackageManager` to find plugins. The benefit here is that the timing is deterministic — you know precisely when you are done with `PackageManager`. However, somehow, you will need to know what is and is not a plugin, in a way that you can determine by information returned from `PackageManager`.

If you happen to know the complete list of possible plugins, you could iterate over that list and use `getPackageInfo()` to see which ones exist and do not exist.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, this reduces your flexibility, as it requires you to know up front the package names of all possible plugins.

Or, you could use `queryIntentActivities()`, `queryIntentServices()`, or `queryBroadcastReceivers()`, providing an `Intent` that identifies some operation a plugin is obligated to implement, to see what matches are found.

There is also a `queryContentProviders()`, but as it does not take an `Intent`, you would have to iterate over each returned `ProviderInfo` to try to determine if it is a `ContentProvider` representing a plugin.

Alternatively, you could call `getInstalledPackages()` on `PackageManager`, to find out about *everything* that is installed, then iterate over them looking for something.

Watching Package-Related Broadcasts

If using `PackageManager` to examine all possible plugins is still too slow, you could optimize things a bit by watching for `ACTION_PACKAGE_ADDED`, `ACTION_PACKAGE_REPLACED`, and `ACTION_PACKAGE_REMOVED` broadcasts, to monitor *changes* to the mix of installed packages. If a known plugin is removed, you can remove it from your roster of installed plugins. When packages are added or replaced, you could use `PackageManager` and `getPackageInfo()` to learn about that specific package, to determine if it represents one of your plugins.

This, however, increases the complexity of your app, as now you need to monitor these broadcasts and maintain your own roster of available plugins somewhere.

Discovery and Usage of the IPC Endpoints

Given that you know that you have a certain number of plugins, represented by a certain set of packages, you can work on actually communicating with them, using any of the available IPC mechanisms. Also, for static data, you have the option of using manifest metadata or well-known resources to publish that data.

No matter what you settle upon, though, you need to consider the impacts of changes to your host app, that might require changes to your interaction with plugins. Everything in this section qualifies as an API that your host app offers to plugins; changes to that API will require you to consider versioning and backwards compatibility.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Component IPC Options

Your plugin could:

- Have an activity, supporting an agreed-upon `Intent` structure, that your app opens as needed
- Have a service, supporting an agreed-upon `Intent` structure, that your app sends commands to or binds to as needed
- Have a `BroadcastReceiver`, supporting an agreed-upon `Intent` structure, that your app can send broadcasts to as needed

For any of those, you would use `setComponentName()` as part of the `Intent`, to specifically identify the plugin that you are talking to.

Your plugin could also have a `ContentProvider` that your host communicates with. That, however, requires that you somehow find out the appropriate authority to use. That authority might be obtained by an agreed-upon algorithm based upon the package name (e.g., the authority is the plugin's package name plus `.PROVIDER`). Or, that authority might be determined by some static data, techniques for which are described in the next section.

In any of these cases, your host's plugin model would document the expectations the host would have of the plugins:

- What `Intent` extras are supported, what their meanings are, and what the data types are for the extras' keys
- What the schema is for the `ContentProvider`
- Etc.

Your host could also be publishing activities, services, receivers, or providers for the plugin to use. So, for example, your host could send a command to a plugin's `IntentService`, that turns around and modifies data in your host's exported `ContentProvider`.

What data is transferred between the host and plugin, of course, is up to you. Bear in mind, though, that IPC cannot handle arbitrary objects. You will need to stick to primitives and basic collections, framework-supplied `Parcelable` classes (e.g., `Bundle`), or your own custom `Parcelable` classes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Static Data Options

Some information that you might need about the plugin is static. In those cases, you do not need to use IPC to get the data, thereby saving the cost of loading the plugin into memory just to invoke some component inside of it.

One option for static data is to use manifest metadata. Any `<activity>`, `<service>`, or `<receiver>` element can have one or more child `<meta-data>` elements. These can hold static data that your host app can read in. There are two major flavors of `<meta-data>` elements:

- A simple key/value pair, where the key is provided by `android:name` and the value is provided by `android:value`
- A key pointing to a resource ID to some other resource, frequently an XML resource (i.e., file in `res/xml/`), providing more details, where the key is in `android:name` and the resource ID is `android:resource`

You will see this approach used in places like app widgets, which use a `<meta-data>` element to point to the app widget metadata, which resides in a separate XML resource.

Your app reads in these values — as literals or identifiers to resources — by retrieving an `ActivityInfo` or `ServiceInfo` object from `PackageManager` for the component (e.g., `getActivityInfo()`, `getReceiverInfo()`, `getServiceInfo()`), then examining the `Bundle` in the `metaData` field of that `...Info` object.

There is nothing stopping you from requiring your plugins implement certain resources or assets in agreed-upon paths. You could then access those resources — or ones from `android:resource` in a `<meta-data>` element — via a `Context` created from `createPackageContext()`. `createPackageContext()` is available on any `Context`, such as an `Activity` or `Service`. Given the package name of your plugin, it gives you a `Context` object that you can use to retrieve resources (`getResources()`) or assets (`getAssets()`) much as you do with one of your own contexts.

Versioning

Any time you are providing programmatic access to your app to others, or any time you are expecting others to provide programmatic access to their apps based upon your specification, you need to bear in mind that your needs may change over time. You may want additional extras, or new bits of static data, or new `Intent` actions. And while you can change your app to take into account your new requirements:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- You have no means of forcing third-party developers to update their apps in lock-step with yours
- You have no means of forcing users to update their plugins and such in lock-step with updating your host app

Hence, you are going to need to deal with versioning your plugin API and supporting older API versions, to offer backwards compatibility for not-yet-updated plugins.

A `<meta-data>` element is perhaps the easiest way to have plugins declare what API version they support. This way, you can find out what "language" the plugin speaks before you try talking to it.

When you then communicate via IPC to the plugins, you will need to take into account what API version the plugin speaks, and adjust your communications accordingly. For example, if you are binding to a plugin's service, you would need to make sure that you are using the right AIDL, to get the right client-side proxy object, one that has the methods and parameters that the plugin supports.

Conversely, if you are providing ways for plugins to initiate communications back to you, you will have to take into account that plugins could be using any outstanding API version. You might elect to use different `Intent` actions or provider authorities to help distinguish the API versions. For example, the plugin sending a command to your service might use `com.suchandso.app.ACTION_PLUGIN.V1` or `com.suchandso.app.ACTION_PLUGIN.V2` in its `Intent`, so you have the flexibility of having a single `Service` handle both of those operations, or splitting them into separate `Service` classes if you feel that will help improve maintainability.

On the whole:

- Be careful in what you send to the plugins. If you claim that certain extras are of certain data types, stick with that, trying to avoid sending other data types that the plugins might not expect.
- Be generous in what you accept from the plugins, particularly where you are changing what you accept from version to version of your API. If you declared that an extra sent to you was originally an `int` and now is a `String`, ideally your new-version code would accept *either* an `int` or a `String`, to help ease the transition.
- Be slow to discontinue support for old API versions. You might use analytics or other data collection mechanisms to get a sense for how many devices are using plugins that speak a particular API version, to give you an idea of how

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

much grief you will get from users if you drop support for that API version and therefore disable certain plugins in an upgrade to your app.

Security

Any time you have inter-process communication, you open up security risks. Hence, intentionally doing IPC means that you intentionally have to consider how best to secure that IPC, to reduce those risks.

Here are three areas of security for you to consider with your plugin model:

User Safe from Permission Leakage

Your plugins, and perhaps the plugin host, may hold various Android permissions, like `READ_CONTACTS` or `INTERNET`. It is incumbent upon you to make sure that either:

- You do not expose information tied to such permissions through your plugin model API (either the host talking to a plugin or vice versa) in a way that other apps could intercept, or
- You ensure that the other party holds the same permission, so that the user knows that the secured information is moving from point to point

For example, suppose that your host app does not hold `READ_CONTACTS`, but a plugin does, specifically to allow the host app to get access to contact information. You need to make sure that, while the host app can get this contact information from the plugin, nobody else can.

Ideally, a plugin developer can be confident that, when the plugin sends information via IPC to the host app, that it is *really* the host app that the plugin is talking to. If some other app can pretend to be the host app, and intercept that information, that other app could potentially use that information to nefarious ends.

Partially, this is an extension of the permission leakage issue described above. It's bad enough that a plugin might leak data to a host app that is not authorized for that data; it is worse if some other app can intercept that data as well.

However, it may be that the data being transferred is not covered by an explicit Android permission, yet might represent information that the user is expecting to keep secure. A financial planner host app using plugins to collect a user's financial data from various banks and brokerages should be taking steps to ensure that the plugin data *only* flows back to the host app, and not to any other apps. This comes

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

despite the fact that Android does not have an `ACCESS_FINANCIAL_DATA` permission as part of the core operating system.

Mostly, this involves having the plugin explicitly state the component that it is communicating with via IPC, rather than relying upon Android derive that information via `Intent` resolution or similar approaches. So, for example, rather than calling `startService()` with just an `Intent` action identifying the host, also set the `ComponentName` on the `Intent` to specifically direct the command to the host app, not to something else advertising that same `Intent` action.

If the host and all its plugins are written by the same firm, you can also use signature-level permissions to restrict access, limiting the IPC to only apps signed by the same signing key.

Host Safe from Trojans

Conversely, if the host app supplies information to the plugins that might represent private or secure data, we need to make sure that the user is comfortable with that data being transferred.

Partially, this involves creating a custom permission that plugins must hold, letting the user know at the time of installing the plugin that this data will be transferred.

Partially, this is making sure that this data is only delivered to the plugins (and, if possible, only to the plugins that specifically need this data). Hence, rather than broadcast `Intents` — even ones where you require a specific permission be held by the receiver — consider using other IPC options that are more "point-to-point", such as sending commands to a specific service identified by its `ComponentName`.

Case Study: DashClock

A Googler's take on an app with a plugin model can be found in DashClock, written by Roman Nurik. DashClock is open source, making it easy to see how he elected to implement his plugin model.

What is DashClock?

Android 4.2 added the notion of lockscreen widgets, app widgets that can go on the lockscreen. DashClock is one such lockscreen widget, designed to replace the standard clock. But, more importantly, it offers a plugin model, so third-party apps

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

can provide dynamic data to be displayed by DashClock, without themselves having to have a lockscreen widget. Similarly, the user can just add DashClock to the lockscreen, not a whole bunch of individual lockscreen widgets.

Discovery… By the User

DashClock helps users find extensions by linking to the Play Store via the following URL: `http://play.google.com/store/search?q=DashClock+Extension&c=apps`.

Anyone publishing a DashClock extension merely needs to describe their app as having (or being) a DashClock extension, and they will automatically show up when the user requests to get more extensions from within DashClock's configuration activity.

DashClock extensions do not *have* to be installed via the Play Store, but DashClock will not directly help improve the "findability" of extensions distributed by other means.

Discovery… By Your App

At its core, DashClock finds extensions by scanning via `PackageManager`. Each extension is obligated to implement a service that advertises an `<action>` of `com.google.android.apps.dashclock.Extension`. DashClock then uses `queryIntentServices()` on `PackageManager` to find these services.

DashClock, however, has the notion of installed versus active extensions. Just because a user installed some app that happens to implement a DashClock extension does not necessarily mean that the user *wants* that app's content cluttering up her DashClock lockscreen widget. Instead, the user not only has to install the app, but tell DashClock to activate that extension. Hence, DashClock has an activity that shows a list of all installed extensions and allows the user to toggle them between active and inactive states (plus order them, etc.).

It is conceivable that the user installs a DashClock extension while this extension-configuration activity is running. Hence, while this activity is running, DashClock registers a `BroadcastReceiver`, via `registerReceiver()`, for the package-management broadcasts (e.g., `ACTION_PACKAGE_ADDED`). Upon receipt of the broadcast, DashClock goes through the original logic to scan using `PackageManager` to find available extensions, then updates the list to match any changes (added extensions, removed extensions, etc.).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

DashClock also monitors for the many of the same broadcasts via a manifest-registered receiver, so it knows when extensions are replaced or removed. In those cases, DashClock needs to determine whether the extension had been active, and if so, what is now required (e.g., removing the extension from the lockscreen widget once it is uninstalled).

Discovery and Usage of the IPC Endpoints

The DashClock app, serving as the plugin host, communicates with its plugins in three main ways:

- Via the aforementioned service, usually implemented as a `DashClockExtension`, which allows DashClock to proactively request that plugins publish updates to their data
- Via an optional "settings activity", which DashClock links to from the extension list, so users can configure the behavior of this specific extension
- Via metadata in the `<service>` element for the `DashClockExtension`

One of the key pieces of metadata is the `protocolVersion`, which tells DashClock what version of the DashClock plugin API the plugin supports.

The plugin turns around and communicates back to DashClock via a service, exported by DashClock under an agreed-upon action. The extension uses this service to publish updates to the data that should be shown for this extension in DashClock's lockscreen widget, much along the lines of how an `AppWidgetProvider` tells the `AppWidgetManager` to update an app widget.

Security

DashClock defines a custom `READ_EXTENSION_DATA` permission. Extensions protect their services by requiring this permission (`android:permission = "com.google.android.apps.dashclock.permission.READ_EXTENSION_DATA"`), so that the user knows about apps seeking to communicate with the extension. Such apps need to hold the `READ_EXTENSION_DATA` permission, meaning that the user will be informed at installation time about the app wishing to speak with DashClock extensions.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Other Plugin Examples

DashClock shows one way of implementing a plugin model, but it is certainly not the only possible implementation. The following sections review some other approaches, to contrast with DashClock's approach.

Plugins by Remote

The biggest challenge with plugins comes at the UI level. While there are many ways to integrate applications for background work (remote services, broadcast Intents, etc.), blending user interfaces is a problem. It is unsafe to have an application execute some plugin's code in its own process, as the plugin may be malicious in nature. Yet, the plugin cannot directly add widgets to the host app's activities any other way.

The key word in that last sentence, of course, is "directly".

There is an indirect way of having one app supply UI components to another app, in the form of the `RemoteViews` object. This is used by app widgets and custom `Notifications`, covered elsewhere in this book.

The plugin can create a `RemoteViews` structure describing the desired UI and deliver that `RemoteViews` to the host app, which can then render that `RemoteViews` wherever it is needed.

This section will outline some of the mechanics behind creating such a UI-centric plugin mechanism.

RemoteViews, Beyond App Widgets

`RemoteViews` are used in a few other places besides app widgets, such as custom `Notification` views. However, you can use `RemoteViews` yourself easily enough. You create one as you would for any other circumstance, like an app widget. To display one, you can use the `apply()` method on the `RemoteViews` object. The `apply()` method takes two parameters:

1. Your `Context`, typically your `Activity`
2. The container into which the contents of the `RemoteViews` will eventually reside

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `apply()` method returns the `View` specified by the rules poured into the `RemoteViews` object... but it does not add it to the container specified in that second parameter. Hence, `apply()` is a bit like calling the three-parameter `inflate()` on a `LayoutInflater` and passing false for the third parameter — you are still responsible for actually adding the `View` to the parent when appropriate.

And that's pretty much it.

Since a `RemoteViews` object implements the `Parcelable` interface, you can store a `RemoteViews` in an `Intent` extra, a `Bundle`, or anything else that works with `Parcelable` (e.g., AIDL-defined remote service interfaces). This is what makes `RemoteViews` so valuable – you can pass one to another process, which can `apply()` it to its own UI.

As a result, `RemoteViews` are a secure way for a plugin to contribute to some host activity's UI. In fact, you can think of an app widget as being a "plugin" for the UI of the home screen.

Thinking About Plugins

So, what does our plugin implementation need?

You have one application (the host) that will be able to display the `RemoteViews` supplied by other applications (the plugins). Somehow, the host will need to know:

1. What plugins are installed
2. How to get `RemoteViews` from the plugins to the host
3. Whether there are plugins that are installed that the user does not want (e.g., app widgets not added to the home screen) or if the user wants to see multiple `RemoteViews` from the same plugin (e.g., multiple instances of an app widget)

As is discussed earlier in this chapter, there are any number of ways of implementing these. The sample shown below will use a broadcast `Intent` to find plugins and another broadcast `Intent` to retrieve `RemoteViews` on demand, while assuming that each plugin will deliver exactly one `RemoteViews`.

Similarly, the plugin will need to know:

1. How it will be activated by the host

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. How it is supposed to deliver `RemoteViews` to the host (broadcast Intent? remote service API? something else?)
3. When it is supposed to deliver `RemoteViews` to the host (pulled by the host? pushed to the host? both?)
4. How many distinct instances of the plugin does the user want (e.g., multiple instances of the app widget), and what is the configuration data for each instance that makes one distinct from the next?

Let's take a look at the `RemoteViews/Host` and `RemoteViews/Plugin` sample applications. These are two apps, each in their own package, implementing a host/plugin relationship, with `RemoteViews` being generated by the plugin and displayed by the host.

In this sample, the plugin will respond to a broadcast `Intent` from the host with a broadcast of its own, signaling that it wishes to serve as a plugin. When the host sends a broadcast to retrieve the `RemoteViews`, the plugin will send a broadcast in response that contains the `RemoteViews`. And, to keep things simple, each plugin will only have one instance (and we will only have one plugin).

Finding Available Plugins

Our host is a simple activity containing a `TextView` as its only content. The expectation is that when the user chooses a Refresh options menu item, we will pull a `RemoteViews` from the plugin and display it.

That, of course, assumes that we have a plugin.

To find plugins, we will send a broadcast, with a custom action, `ACTION_CALL_FOR_PLUGINS`. Any plugin implementation would need a `BroadcastReceiver` set up in the manifest to respond to such an action.

To keep things simple, the host will only have one plugin. The plugin itself will be represented by a `ComponentName` object, identifying the implementation of the plugin, held in a `pluginCN` data member:

```
  private ComponentName pluginCN=null;
```

In `onResume()`, if we do not have a plugin yet, we send the broadcast to try to find one:

```
  @Override
  public void onResume() {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onResume();

    IntentFilter pluginFilter=new IntentFilter();

    pluginFilter.addAction(ACTION_REGISTER_PLUGIN);
    pluginFilter.addAction(ACTION_DELIVER_CONTENT);

    registerReceiver(plugin, pluginFilter, PERM_ACT_AS_PLUGIN, null);

    if (pluginCN == null) {
      sendBroadcast(new Intent(ACTION_CALL_FOR_PLUGINS));
    }
  }
```

Responding to the Call for Plugins

Over in our plugin implementation, we do indeed have a `BroadcastReceiver` — cunningly named `Plugin` — with a manifest entry set up to respond to our `ACTION_CALL_FOR_PLUGINS` broadcast.

What the host wants in response is to receive a broadcast from the plugin, with an action of `ACTION_REGISTER_PLUGIN`, and an extra of `EXTRA_COMPONENT`, containing the `ComponentName` of the `BroadcastReceiver` that is the plugin implementation. So, when `Plugin` receives an `ACTION_CALL_FOR_PLUGINS` broadcast, it does just that:

```
package com.commonsware.android.rv.plugin;

import android.content.BroadcastReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.widget.RemoteViews;

public class Plugin extends BroadcastReceiver {
  public static final String ACTION_CALL_FOR_PLUGINS=
      "com.commonsware.android.rv.host.CALL_FOR_PLUGINS";
  public static final String ACTION_REGISTER_PLUGIN=
      "com.commonsware.android.rv.host.REGISTER_PLUGIN";
  public static final String ACTION_CALL_FOR_CONTENT=
      "com.commonsware.android.rv.host.CALL_FOR_CONTENT";
  public static final String ACTION_DELIVER_CONTENT=
      "com.commonsware.android.rv.host.DELIVER_CONTENT";
  public static final String EXTRA_COMPONENT="component";
  public static final String EXTRA_CONTENT="content";
  private static final String HOST_PACKAGE="com.commonsware.android.rv.host";

  @Override
  public void onReceive(Context ctxt, Intent i) {
    if (ACTION_CALL_FOR_PLUGINS.equals(i.getAction())) {
      Intent registration=new Intent(ACTION_REGISTER_PLUGIN);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      registration.setPackage(HOST_PACKAGE);
      registration.putExtra(EXTRA_COMPONENT,
                            new ComponentName(ctxt, getClass()));

      ctxt.sendBroadcast(registration);
    }
    else if (ACTION_CALL_FOR_CONTENT.equals(i.getAction())) {
      RemoteViews rv=
          new RemoteViews(ctxt.getPackageName(), R.layout.plugin);
      Intent update=new Intent(ACTION_DELIVER_CONTENT);

      update.setPackage(HOST_PACKAGE);
      update.putExtra(EXTRA_CONTENT, rv);
      ctxt.sendBroadcast(update);
    }
  }
}
```

For added security, we use `setPackage()` in the plugin, so the `ACTION_REGISTER_PLUGIN` broadcast can only be received by the host.

The host activity needs to receive `ACTION_REGISTER_PLUGIN` broadcasts. Hence, it has a `BroadcastReceiver` implementation, in the `plugin` data member, that it registers for `ACTION_REGISTER_PLUGIN` in `onResume()`. The `plugin` `BroadcastReceiver`, upon receiving an `ACTION_REGISTER_PLUGIN` broadcast, grabs the `ComponentName` out of the `EXTRA_COMPONENT` extra and stores it in `pluginCN`:

```
  private BroadcastReceiver plugin=new BroadcastReceiver() {
    @Override
    public void onReceive(Context ctxt, Intent i) {
      if (ACTION_REGISTER_PLUGIN.equals(i.getAction())) {
        pluginCN=(ComponentName)i.getParcelableExtra(EXTRA_COMPONENT);
      }
      else if (ACTION_DELIVER_CONTENT.equals(i.getAction())) {
        RemoteViews rv=(RemoteViews)i.getParcelableExtra(EXTRA_CONTENT);
        ViewGroup frame=(ViewGroup)findViewById(android.R.id.content);

        frame.removeAllViews();

        View pluginView=rv.apply(RemoteViewsHostActivity.this, frame);

        frame.addView(pluginView);
      }
    }
  };
```

At this point, we wait for the user to click the Refresh options menu item.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Requesting RemoteViews

When the user does indeed choose Refresh, we call a `refreshPlugin()` method on the host activity:

```
private void refreshPlugin() {
  Intent call=new Intent(ACTION_CALL_FOR_CONTENT);

  call.setComponent(pluginCN);
  sendBroadcast(call);
}
```

Here, we send an `ACTION_CALL_FOR_CONTENT` broadcast, with the target component set to be the plugin implementation, as identified by its `ComponentName`. This ensures that this broadcast will only go to that plugin app and nobody else.

Responding with RemoteViews

Our `Plugin` is also registered in the manifest to respond to `ACTION_CALL_FOR_CONTENT`. So, when that broadcast arrives, it can create the `RemoteViews` in response, sending it out via an `ACTION_DELIVER_CONTENT` broadcast back to the host. Once again, we use `setPackage()` to restrict the broadcast to be the host's package. The broadcast also has the `RemoteViews` tucked in an `EXTRA_CONTENT` extra.

Our host activity registered the `plugin` `BroadcastReceiver` for `ACTION_DELIVER_CONTENT` as well. So, when that broadcast arrives, it can utilize the `RemoteViews`. We find the `ViewGroup` that is the root of our content (`android.R.id.content`), wipe out whatever is in it now, `apply()` the `RemoteViews` to that `ViewGroup`, and add the resulting `View` to the `ViewGroup`. This has the net effect of getting rid of our original `TextView` content, replacing it with whatever the plugin poured into the `RemoteViews`. Or, if the user chooses Refresh again, the older `RemoteViews`-generated content is replaced with fresh content.

Dealing with Android 3.1+

To test this, install the Host application, followed by the Plugin application. On Android 3.0 and older, running the Host and choosing the Refresh options menu item will change the display from its original state to the one with the plugin's `RemoteViews`.

However, that will not work right away on Android 3.1 and higher.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

On these versions of Android, applications are installed into a "stopped" state, where no `BroadcastReceiver` in the manifest will work, until the user manually runs the application. The simplest way to do that is via an activity. So, the Plugin project has a trivial activity that just displays a `Toast` and exits:

```
package com.commonsware.android.rv.plugin;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Toast;

public class PluginActivationActivity extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    Toast.makeText(this, R.string.activated, Toast.LENGTH_LONG).show();
    finish();
  }
}
```

You will need to run this activity on Android 3.1 and higher first, then run the Host project's activity, to get the plugin to work.

If you happen to install these on an Android 3.0 or older device, though, you may wonder if the author has lost his marbles. That is because you will not see any activity associated with the Plugin application.

Since the author has not owned marbles in a few decades, clearly there must be some other answer. In this case, we use a variation of a trick pointed out by Daniel Lew.

Our `<activity>` element in the manifest has an `android:enabled` attribute. A disabled activity does not show up in the launcher. But rather than have `android:enabled` specifically tied to `true` or `false` in the manifest, it references a boolean resource:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.rv.plugin"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk android:minSdkVersion="7"/>

  <uses-permission android:name="com.commonsware.android.rv.host.ACT_AS_PLUGIN"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <receiver
      android:name="Plugin"
      android:permission="com.commonsware.android.rv.host.ACT_AS_HOST">
      <intent-filter>
        <action android:name="com.commonsware.android.rv.host.CALL_FOR_PLUGINS"/>
        <action android:name="com.commonsware.android.rv.host.CALL_FOR_CONTENT"/>
      </intent-filter>
    </receiver>

    <activity
      android:name="PluginActivationActivity"
      android:enabled="@bool/i_has_needs_activity"
      android:excludeFromRecents="true"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

In `res/values/bools.xml`, we define that boolean resource to be `false`, meaning the activity will not appear in the launcher:

```
<resources>

  <bool name="i_has_needs_activity">false</bool>

</resources>
```

But, in `res/values-v12/bools.xml`, we define that boolean resource to be `true`, causing the activity to appear on Android 3.1 and higher:

```
<resources>

  <bool name="i_has_needs_activity">true</bool>

</resources>
```

This way, our extraneous activity does not clutter up older devices where it is not needed. Mr. Lew's blog post on this subject points out that this trick can be used to have different implementations of an app widget for different Android versions (e.g., one that uses a `ListView` for API Level 11 and higher, plus one that does not for older devices).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Permission Scheme

Another thing that these sample projects use are custom permissions, to help with security.

To serve as a plugin host, you must hold the `ACTS_AS_HOST` permission. To serve as a plugin implementation, you must hold the `ACTS_AS_PLUGIN` permission. These are defined in the Host project's manifest:

```
  <permission
    android:name="com.commonsware.android.rv.host.ACT_AS_HOST"
    android:description="@string/host_desc"
    android:label="@string/host_label">
  </permission>
  <permission
    android:name="com.commonsware.android.rv.host.ACT_AS_PLUGIN"
    android:description="@string/plugin_desc"
    android:label="@string/plugin_label">
  </permission>
```

Each application then has its appropriate `<uses-permission>` element for the role that it plays, such as the Plugin holding the `ACTS_AS_PLUGIN` permission:

```
  <uses-permission android:name="com.commonsware.android.rv.host.ACT_AS_PLUGIN"/>
```

The `BroadcastReceiver` defined by the Plugin project has, in its `<receiver>` element, the `android:permission` attribute, indicating that whoever sends a broadcast to this receiver must holds `ACTS_AS_HOST`:

```
    <receiver
      android:name="Plugin"
      android:permission="com.commonsware.android.rv.host.ACT_AS_HOST">
      <intent-filter>
        <action android:name="com.commonsware.android.rv.host.CALL_FOR_PLUGINS"/>
        <action android:name="com.commonsware.android.rv.host.CALL_FOR_CONTENT"/>
      </intent-filter>
    </receiver>
```

Similarly, the `BroadcastReceiver` defined dynamically by the host activity uses a version of `registerReceiver()` that takes the permission the sender must hold:

```
    registerReceiver(plugin, pluginFilter, PERM_ACT_AS_PLUGIN, null);
```

That permission is defined in a static data member:

```
  public static final String PERM_ACT_AS_PLUGIN=
      "com.commonsware.android.rv.host.ACT_AS_PLUGIN";
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This way, the user is informed about the host/plugin relationship and can make appropriate decisions when they install plugins.

Note, though, that for this to work, the host application must be installed first, to define the custom permissions. If a plugin is installed before the host, there is no error, but the plugin will not be granted the as-yet-undefined custom permissions, and so the plugin will not work. The user would have to uninstall and reinstall the plugin after installing the host to fix this problem.

Other Plugin Features and Issues

It is possible for the `apply()` method on `RemoteViews` to throw a `RuntimeException`. For example, the `RemoteViews` might contain a reference to a widget ID that does not exist within the inflated views of the `RemoteViews` itself. Since `apply()` does not throw a checked exception, it is easy to do what we did in the sample app and assume `apply()` will succeed, but it very well may not. A robust implementation of this plugin system would wrap the `apply()` call in an exception handler that would do something useful if the plugin's `RemoteViews` has a bug.

You need to be a bit careful to make sure that a plugin can only update itself. The sample app assumes that the only thing that will send an `ACTION_DELIVER_CONTENT` broadcast to it will be the plugin, but that is not necessarily the case. In principle, anything that holds the `ACTS_AS_PLUGIN` permission could send an `ACTION_DELIVER_CONTENT` to the host, and thereby specify what the `RemoteViews` are. A robust plugin system would have some sort of shared secret, such as an identifier, between the host and the plugin, so another component cannot readily masquerade as being the plugin itself.

ContentProvider Plugins

Another way to extend your application at runtime is via plugins implemented via the `ContentProvider` framework. You could create new `ContentProvider` implementations that offer up data, perhaps using a consistent schema. Then, you could find those providers via a naming convention (e.g., for a main application with a package of `com.foo.abc`, your plugin apps would be `com.foo.abc.plugin.*`) and `PackageManager`, perhaps using a provider `Uri` naming convention to allow the host to know how to query the plugin.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, there are other ways of employing a `ContentProvider` to help as a plugin, and this section explores one specific scenario: reducing the host app's permission requirements.

The Problem: Permission Creep

At the moment, for standard versions of Android, apps cannot request "conditional" or "optional" permissions, that the user could elect to opt out of. Instead, apps must request in their manifest all possible permissions that they could need. This is considered by many to be a significant limitation, but Google has stated repeatedly that they are not considering alternative strategies.

The net effect, though, is that an app often times needs a lot of permissions, or needs to add new permissions (requiring existing users to agree to the new permission list). Such lists of permissions can dissuade potential users from installing the app in the first place.

However, even though Android does not provide a simple and clean way for users to opt into (or out of) certain permissions for certain apps, plugins can offer a similar model. The base app can require some permissions for some features, with other features (and their respective permissions) added via plugins. Users can elect to install the plugins and agree to those permissions, or abandon or never install the plugins in the first place.

The hassle, of course, is in implementing the plugin APK and connecting to it from the main app. The plugin needs to have all the functionality that must directly use classes and methods secured by the permission. This can increase the complexity in maintaining the overall app.

A Solution: ContentProvider Proxies

Some permissions exist primarily to protect a `ContentProvider`, such as `READ_CONTACTS` and `WRITE_CONTACTS` for the `ContactsContract` provider.

The nice thing about the `ContentProvider` framework is that it is simply a contract. You use a `ContentResolver` and some magic values (`Uri`, "projection" of columns to return, etc.), and you get results. In fact, you can even change some of those magic values – any `Uri` supporting the same columns could be used with all the same client Java code, just by changing the `Uri` itself.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That allows us to create a proxy for `ContentProvider`. The proxy APK will hold the permission and call the real `ContentProvider` as needed. The proxy APK will expose its own `ContentProvider`, with a different `Uri`. Done properly — such that only the host app can use the proxy — the proxy will isolate the permission(s) for the real `ContentProvider` in the plugin. A `ContactsContract` proxy, for example, could hold `READ_CONTACTS` and `WRITE_CONTACTS`, proxying requests on behalf of a main app that lacks those permissions.

To secure the proxy, we need to ensure that only our apps can use the proxy, not anyone else's apps. Otherwise, those third-party apps could get at, say, contacts without the `READ_CONTACTS` permission.

The simplest way to accomplish this is to use a signature-level custom permission.

Any app can declare a new permission via the `<permission>` element in the manifest. Normally, any app can request to hold this permission via `<uses-permission>`, and the user will be able to grant or deny this request at install time, just like any system-defined permission.

However, it is possible to add an `android:protectionLevel="signature"` attribute to the `<permission>` element. In this case, only apps signed by the same signing key will be able to request the permission — everyone else is automatically denied. Furthermore, apps signed by the same signing key will automatically get the permission without the user having to approve it.

So, you can have the proxy require a signature-level custom permission, thereby limiting possible consumers of the proxy to be signed by the same signing key.

Let's look at a pair of projects that create and consume a proxy for the `CallLog` `ContentProvider`. These projects are located in the `Introspection/CPProxy` directory and are named `Provider` and `Consumer`, respectively.

Note that this sample works only on API Level 11 and higher, due to the consumer's use of the native implementation of the `Loader` framework.

Provider

Most of the logic for our provider proxy can be found in the `AbstractCPProxy` base class. It implements the mandatory methods for the `ContentProvider` contract — such as `insert()` — and simply turns around and forwards those requests along to another provider:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
                    String[] selectionArgs, String sortOrder) {
  checkTainted();

  Cursor result=
      getContext().getContentResolver().query(convertUri(uri),
                                              projection, selection,
                                              selectionArgs,
                                              sortOrder);

  return(new CrossProcessCursorWrapper(result));
}

@Override
public Uri insert(Uri uri, ContentValues values) {
  checkTainted();

  return(getContext().getContentResolver().insert(convertUri(uri),
                                                  values));
}

@Override
public int update(Uri uri, ContentValues values, String selection,
                  String[] selectionArgs) {
  checkTainted();

  return(getContext().getContentResolver().update(convertUri(uri),
                                                  values, selection,
                                                  selectionArgs));
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
  checkTainted();

  return(getContext().getContentResolver().delete(convertUri(uri),
                                                  selection,
                                                  selectionArgs));
}

@Override
public String getType(Uri uri) {
  checkTainted();

  return(getContext().getContentResolver().getType(convertUri(uri)));
}
```

The `checkTainted()` calls are part of our confirming that our custom permission is OK, and that is covered in the chapter on advanced permissions. For the purposes of this chapter, just ignore them (along with the `onCreate()` method not shown here).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It is up to a subclass of `AbstractCPProxy` to implement the `convertUri()` method, which takes the `Uri` supplied by the consumer and transforms it into the proper `Uri` to use for making the real request. In this case, our subclass is `CallLogProxy`:

```
package com.commonsware.android.cpproxy.provider;

import android.content.ContentUris;
import android.net.Uri;
import android.provider.CallLog;

public class CallLogProxy extends AbstractCPProxy {
  protected Uri convertUri(Uri uri) {
    long id=ContentUris.parseId(uri);

    if (id >= 0) {
      return(ContentUris.withAppendedId(CallLog.Calls.CONTENT_URI, id));
    }

    return(CallLog.Calls.CONTENT_URI);
  }
}
```

Here, we grab the instance ID off the end of the `Uri` (if it exists) and generate a new `Uri` based on `CallLog.CONTENT_URI`, indicating that we want to forward our requests to the `CallLog`.

The biggest complexity of the standard CRUD `ContentProvider` methods comes with `query()`. The `Cursor` returned by `query()` must implement the `CrossProcessCursor` interface. The `SQLiteCursor` implementation supports this interface, which is why typical providers do not worry about this requirement. However, the `Cursor` returned by `query()` on `ContentResolver` is not necessarily a `CrossProcessCursor`. Hence, we need to wrap it in a `CursorWrapper` that does implement `CrossProcessCursor`:

```
  private static final String PREFS_TAINTED="tainted";
  private boolean tainted=false;

  public AbstractCPProxy() {
    super();
  }

  @Override
  public boolean onCreate() {
    SharedPreferences prefs=
        PreferenceManager.getDefaultSharedPreferences(getContext());
```

The resulting `CrossProcessCursorWrapper`, as originally shown in a Stack Overflow answer, looks like this:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  checkTainted();

  Cursor result=
      getContext().getContentResolver().query(convertUri(uri),
                                              projection, selection,
                                              selectionArgs,
                                              sortOrder);

  return(new CrossProcessCursorWrapper(result));
}

@Override
public Uri insert(Uri uri, ContentValues values) {
  checkTainted();

  return(getContext().getContentResolver().insert(convertUri(uri),
                                                  values));
}

@Override
public int update(Uri uri, ContentValues values, String selection,
                  String[] selectionArgs) {
  checkTainted();

  return(getContext().getContentResolver().update(convertUri(uri),
                                                  values, selection,
                                                  selectionArgs));
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
  checkTainted();

  return(getContext().getContentResolver().delete(convertUri(uri),
                                                  selection,
                                                  selectionArgs));
}

@Override
public String getType(Uri uri) {
  checkTainted();

  return(getContext().getContentResolver().getType(convertUri(uri)));
}

private void checkTainted() {
  if (tainted) {
    throw new RuntimeException(getContext().getString(R.string.tainted_abort));
  }
}

// following from
// http://stackoverflow.com/a/5243978/115145
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  public class CrossProcessCursorWrapper extends CursorWrapper
      implements CrossProcessCursor {
    public CrossProcessCursorWrapper(Cursor cursor) {
```

Note that this implementation has been largely untested by this book's author, though it appears to work.

The manifest for this project has three items of note:

- It has the `<uses-permission>` element for `READ_CONTACTS`, while our consumer project will not
- It has a `<permission>` element, defining a custom `com.commonsware.android.cpproxy.PLUGIN` permission that has signature-level protection
- It has our `<provider>`, requiring that custom permission, and declaring its authority to be `com.commonsware.android.cpproxy.CALL_LOG`

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.cpproxy.provider"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="9"
    android:targetSdkVersion="11"/>

  <uses-permission android:name="android.permission.READ_CONTACTS"/>

  <permission
    android:name="com.commonsware.android.cpproxy.PLUGIN"
    android:protectionLevel="signature">
  </permission>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <provider
      android:name=".CallLogProxy"
      android:authorities="com.commonsware.android.cpproxy.CALL_LOG"
      android:permission="com.commonsware.android.cpproxy.PLUGIN">
    </provider>
  </application>

</manifest>
```

Note that a complete `AbstractCPProxy` implementation should forward along all the other methods as well (e.g., `call()`).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Consumer

Our Consumer project is nearly identical to the `CalendarContract` sample from elsewhere in this book.

However, instead of the `READ_CONTACTS` permission, we declare that we need the `com.commonsware.android.cpproxy.PLUGIN` permission instead:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.cpproxy.consumer"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="11"
    android:targetSdkVersion="11"/>

  <uses-permission android:name="com.commonsware.android.cpproxy.PLUGIN"/>

  <permission
    android:name="com.commonsware.android.cpproxy.PLUGIN"
    android:protectionLevel="signature">
  </permission>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name=".CPProxyConsumerActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Also, our `CONTENT_URI` is no longer the one found on `CallLog`, but rather one identifying our proxy:

```
  private static final Uri CONTENT_URI=
      Uri.parse("content://com.commonsware.android.cpproxy.CALL_LOG");
```

And there are minor changes because we are querying `CallLog` (indirectly) rather than `CalendarContract`, such as a change in the columns for our projection:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  private static final String[] PROJECTION=new String[] {
      CallLog.Calls._ID, CallLog.Calls.NUMBER, CallLog.Calls.DATE };
```

Otherwise, the consumer projects are the same. The difference is that our consumer project does not need the `READ_CONTACTS` permission the same way that the original needed the `READ_CALENDAR` permission.

In this case, the consumer project depends entirely upon the existence of the plugin — otherwise, the consumer project has no value. Hence, in this case, going the plugin route is silly. But an application that could use the `CallLog` but does not depend upon it could use this approach to isolate the `READ_CONTACTS` requirement in a plugin, so users could elect to install the plugin or not, and the main app would not need to request `READ_CONTACTS` and add to the roster of permissions the user must agree to up front.

Note that, in principle, the consumer should contain some of the same defenses against custom permission changes that the proxy does (in the form of those `checkTainted()` calls). This is covered in greater detail in the chapter on advanced permissions.

Limitations of the Approach

There will be additional overhead in using the proxy, which will hamper performance. Ideally, this plugin mechanism is only used for features that need light use of the protected `ContentProvider`, so the overhead will not be a burden to the user.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

PackageManager Tricks

`PackageManager` is your primary means of introspection at the component level, to determine what else is installed on the device and what components they export (activities, etc.). As such, there are many ways you can use `PackageManager` to determine if something you want is possible or not, so you can modify your behavior accordingly (e.g., disable action bar items that are not possible).

This chapter will outline some ways you can use `PackageManager` to find out what components are available to you on a device.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Asking Around

The ways to find out whether there is an activity that will respond to a given `Intent` are by means of `queryIntentActivityOptions()` and the somewhat simpler `queryIntentActivities()`.

The `queryIntentActivityOptions()` method takes the caller `ComponentName`, the "specifics" array of `Intent` instances, the overall `Intent` representing the actions you are seeking, and the set of flags. It returns a `List` of `Intent` instances matching the stated criteria, with the "specifics" ones first.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you would like to offer alternative actions to users, but by means other than `addIntentOptions()`, you could call `queryIntentActivityOptions()`, get the `Intent` instances, then use them to populate some other user interface (e.g., a toolbar).

A simpler version of this method, `queryIntentActivities()`, is used by the `Introspection/Launchalot` sample application. This presents a "launcher" — an activity that starts other activities — but uses a `ListView` rather than a grid like the Android default home screen uses.

Here is the Java code for `Launchalot` itself:

```
package com.commonsware.android.launchalot;

import android.app.ListActivity;
import android.content.ComponentName;
import android.content.Intent;
import android.content.pm.ActivityInfo;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.TextView;
import java.util.Collections;
import java.util.List;

public class Launchalot extends ListActivity {
  AppAdapter adapter=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    PackageManager pm=getPackageManager();
    Intent main=new Intent(Intent.ACTION_MAIN, null);

    main.addCategory(Intent.CATEGORY_LAUNCHER);

    List<ResolveInfo> launchables=pm.queryIntentActivities(main, 0);

    Collections.sort(launchables,
                     new ResolveInfo.DisplayNameComparator(pm));

    adapter=new AppAdapter(pm, launchables);
    setListAdapter(adapter);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  protected void onListItemClick(ListView l, View v,
                                 int position, long id) {
    ResolveInfo launchable=adapter.getItem(position);
    ActivityInfo activity=launchable.activityInfo;
    ComponentName name=new ComponentName(activity.applicationInfo.packageName,
                                         activity.name);
    Intent i=new Intent(Intent.ACTION_MAIN);

    i.addCategory(Intent.CATEGORY_LAUNCHER);
    i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
               Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED);
    i.setComponent(name);

    startActivity(i);
  }

  class AppAdapter extends ArrayAdapter<ResolveInfo> {
    private PackageManager pm=null;

    AppAdapter(PackageManager pm, List<ResolveInfo> apps) {
      super(Launchalot.this, R.layout.row, apps);
      this.pm=pm;
    }

    @Override
    public View getView(int position, View convertView,
                        ViewGroup parent) {
      if (convertView==null) {
        convertView=newView(parent);
      }

      bindView(position, convertView);

      return(convertView);
    }

    private View newView(ViewGroup parent) {
      return(getLayoutInflater().inflate(R.layout.row, parent, false));
    }

    private void bindView(int position, View row) {
      TextView label=(TextView)row.findViewById(R.id.label);

      label.setText(getItem(position).loadLabel(pm));

      ImageView icon=(ImageView)row.findViewById(R.id.icon);

      icon.setImageDrawable(getItem(position).loadIcon(pm));
    }
  }
}
```

In `onCreate()`, we:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. Get a `PackageManager` object via `getPackageManager()`
2. Create an `Intent` for `ACTION_MAIN` in `CATEGORY_LAUNCHER`, which identifies activities that wish to be considered "launchable"
3. Call `queryIntentActivities()` to get a `List` of `ResolveInfo` objects, each one representing one launchable activity
4. Sort those `ResolveInfo` objects via a `ResolveInfo.DisplayNameComparator` instance
5. Pour them into a custom `AppAdapter` and set that to be the contents of our `ListView`

`AppAdapter` is an `ArrayAdapter` subclass that maps the icon and name of the launchable `Activity` to a row in the `ListView`, using a custom row layout.

Finally, in `onListItemClick()`, we construct an `Intent` that will launch the clicked-upon `Activity`, given the information from the corresponding `ResolveInfo` object. Not only do we need to populate the `Intent` with `ACTION_MAIN` and `CATEGORY_LAUNCHER`, but we also need to set the component to be the desired `Activity`. We also set `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED` flags, following Android's own launcher implementation from the Home sample project. Finally, we call `startActivity()` with that `Intent`, which opens up the activity selected by the user.

The result is a simple list of launchable activities:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

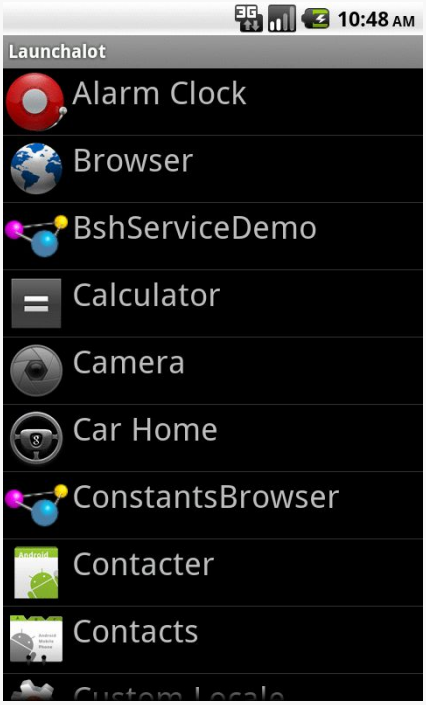


Figure 689: The Launchalot sample application

There is also a `resolveActivity()` method that takes a template Intent, as do `queryIntentActivities()` and `queryIntentActivityOptions()`. However, `resolveActivity()` returns the single best match, rather than a list.

NOTE: On modern versions of Android, there is a `LauncherApps` class that simplifies a lot of this and takes things like Android Work profiles into account. For really implementing a home screen-style launcher, you will probably want to use `LauncherApps`. However, using `PackageManager` to find what can handle certain `Intent` structures is used for other purposes beyond home screen launchers.

Preferred Activities

Users, when presented with a default activity chooser, usually have the option to check a `CheckBox` indicating that they want to make their next choice be the default for this action for now on. The next time they do whatever they did to bring up the chooser, it should go straight to this default. This is known in the system as the "preferred activity" for an `Intent` structure, and is stored in the system as a set of pairs of `IntentFilter` objects and the corresponding `ComponentName` of the preferred activity.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To find out what the preferred activities are on a given device, you can ask `PackageManager` to `getPreferredActivities()`. You pass in a `List<IntentFilter>` and a `List<ComponentName>`, and Android fills in those lists with the preferred activity information.

To see this in action, take a look at the `Introspection/PrefActivities` sample application. This simply loads all of the information into a `ListView`, using `android.R.layout.simple_list_item_2` as a row layout for a title-and-description pattern.

The `PackageManager` logic is confined to `onCreate()`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  PackageManager mgr=getPackageManager();

  mgr.getPreferredActivities(filters, names, null);
  setListAdapter(new IntentFilterAdapter());
}
```

In this case, the two lists are data members of the activity:

```
ArrayList<IntentFilter> filters=new ArrayList<IntentFilter>();
ArrayList<ComponentName> names=new ArrayList<ComponentName>();
```

Most of the logic is in formatting the `ListView` contents. `IntentFilter`, unfortunately, does not come with a method that gives us a human-readable dump of its definition. As a result, we need to roll that ourselves. Compounding the problem is that `IntentFilter` tends to return `Iterator` objects for its collections (e.g., roster of actions), rather than something `Iterable`. The activity leverages an `Iterator`-to-`Iterable` wrapper culled from a Stack Overflow answer to help with this. The `IntentFilterAdapter` and helper code looks like this:

```
// from http://stackoverflow.com/a/8555153/115145

public static <T> Iterable<T> in(final Iterator<T> iterator) {
  class SingleUseIterable implements Iterable<T> {
    private boolean used=false;

    @Override
    public Iterator<T> iterator() {
      if (used) {
        throw new IllegalStateException("Already invoked");
      }
      used=true;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      return iterator;
    }
  }
  return new SingleUseIterable();
}

class IntentFilterAdapter extends ArrayAdapter<IntentFilter> {
  IntentFilterAdapter() {
    super(PreferredActivitiesDemoActivity.this,
          android.R.layout.simple_list_item_2, android.R.id.text1,
          filters);
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    View row=super.getView(position, convertView, parent);
    TextView filter=(TextView)row.findViewById(android.R.id.text1);
    TextView name=(TextView)row.findViewById(android.R.id.text2);

    filter.setText(buildTitle(getItem(position)));
    name.setText(names.get(position).getClassName());

    return(row);
  }

  String buildTitle(IntentFilter filter) {
    StringBuilder buf=new StringBuilder();
    boolean first=true;

    if (filter.countActions() > 0) {
      for (String action : in(filter.actionsIterator())) {
        if (first) {
          first=false;
        }
        else {
          buf.append('/');
        }

        buf.append(action.replaceAll("android.intent.action.", ""));
      }
    }

    if (filter.countDataTypes() > 0) {
      first=true;

      for (String type : in(filter.typesIterator())) {
        if (first) {
          buf.append(" : ");
          first=false;
        }
        else {
          buf.append('|');
        }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
          buf.append(type);
        }
      }

      if (filter.countDataSchemes() > 0) {
        buf.append(" : ");
        buf.append(filter.getDataScheme(0));

        if (filter.countDataSchemes() > 1) {
          buf.append(" (other schemes)");
        }
      }

      if (filter.countDataPaths() > 0) {
        buf.append(" : ");
        buf.append(filter.getDataPath(0));

        if (filter.countDataPaths() > 1) {
          buf.append(" (other paths)");
        }
      }

      return(buf.toString());
    }
  }
```

The resulting activity shows a simple description of the `IntentFilter` along with the class name of the corresponding activity in each row:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

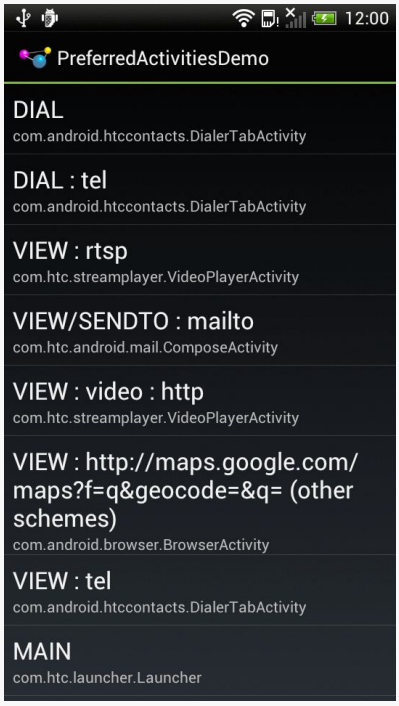


Figure 690: Preferred Activities on a Stock HTC One S

Another way to think about preferred activities is to determine what specific activity will handle a `startActivity()` call on some `Intent`. If there is only one alternative, or the user chose a preferred activity, that activity should handle the `Intent`. Otherwise, the activity handling the `Intent` *should* be one implementing a chooser. The `resolveActivity()` method on `PackageManager` can let us know what will handle the `Intent`.

To examine what `resolveActivity()` returns, take a look at the `Introspection/Resolver` sample application.

The activity — which uses `Theme.NoDisplay` and so has no UI of its own — is fairly short:

```
package com.commonsware.android.resolver;

import android.app.Activity;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;
import android.net.Uri;
import android.os.Bundle;
import android.widget.Toast;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public class ResolveActivityDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    PackageManager mgr=getPackageManager();
    Intent i=
        new Intent(Intent.ACTION_VIEW,
                   Uri.parse("http://commonsware.com"));
    ResolveInfo ri=
        mgr.resolveActivity(i, PackageManager.MATCH_DEFAULT_ONLY);

    Toast.makeText(this, ri.loadLabel(mgr), Toast.LENGTH_LONG).show();

    startActivity(i);
    finish();
  }
}
```

We get a `PackageManager`, create an `Intent` to test, and pass the `Intent` to `resolveActivity()`. We include `MATCH_DEFAULT_ONLY` so we only get activities that have `CATEGORY_DEFAULT` in their `<intent-filter>` elements. We then use `loadLabel()` on the resulting `ResolveInfo` object to get the display name of the activity, toss that in a `Toast`, and invoke `startActivity()` on the `Intent` to confirm the results.

On a device with only one option, or with a default chosen, the `Toast` will show the name of the preferred activity (e.g., Browser). On most devices with more than one option, the `startActivity()` call will display a chooser, and the `Toast` will show the display name of the chooser (e.g., "Android System").

However, on some devices — notably newer models from HTC distributed in the US — `resolveActivity()` indicates that `HTCLinkifyDispatcher` is the one that will handle `ACTION_VIEW` on a URL... even if there is more than one browser installed and no default has been specified. This is part of a workaround that HTC added in 2012 to help deal with a patent dispute with Apple.

Middle Management

The `PackageManager` class offers much more than merely `queryIntentActivities()` and `queryIntentActivityOptions()`. It is your gateway to all sorts of analysis of what is installed and available on the device where your application is installed and available. If you want to be able to intelligently connect to third-party applications based on whether or not they are around, `PackageManager` is what you will want.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Finding Applications and Packages

Packages are what get installed on the device — a package is the in-device representation of an APK. An application is defined within a package's manifest. Between the two, you can find out all sorts of things about existing software installed on the device.

Specifically, `getInstalledPackages()` returns a `List` of `PackageInfo` objects, each of which describes a single package. Here, you can find out:

1. The version of the package, in terms of a monotonically increasing number (`versionCode`) and the display name (`versionName`)
2. Details about all of the components — activities, services, etc. — offered by this package
3. Details about the permissions the package requires

Similarly, `getInstalledApplications()` returns a `List` of `ApplicationInfo` objects, each providing data like:

1. The user ID that the application will run as
2. The path to the application's private data directory
3. Whether or not the application is enabled

In addition to those methods, you can call:

1. `getApplicationIcon()` and `getApplicationLabel()` to get the icon and display name for an application
2. `getLaunchIntentForPackage()` to get an `Intent` for something launchable within a named package
3. `setApplicationEnabledSetting()` to enable or disable an application

Finding Resources

You can access resources from another application, apparently without any security restrictions. This may be useful if you have multiple applications and wish to share resources for one reason or another.

The `getResourcesForActivity()` and `getResourcesForApplication()` methods on `PackageManager` return a `Resources` object. This is just like the one you get for your own application via `getResources()` on any `Context` (e.g., `Activity`). However, in

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

this case, you identify what activity or application you wish to get the `Resources` from (e.g., supply the application's package name as a `String`).

There are also `getText()` and `getXml()` methods that dive into the `Resources` object for an application and pull out specific `String` or `XmlPullParser` objects. However, these require you to know the resource ID of the resource to be retrieved, and that may be difficult to manage between disparate applications.

Finding Components

Not only does Android offer "query" and "resolve" methods to find activities, but it offers similar methods to find other sorts of Android components:

1. `queryBroadcastReceivers()`
2. `queryContentProviders()`
3. `queryIntentServices()`
4. `resolveContentProvider()`
5. `resolveService()`

For example, you could use `resolveService()` to determine if a certain remote service is available, so you can disable certain UI elements if the service is not on the device. You could achieve the same end by calling `bindService()` and watching for a failure, but that may be later in the application flow than you would like.

There is also a `setComponentEnabledSetting()` to toggle a component (activity, service, etc.) on and off. While this may seem esoteric, there are a number of possible uses for this method, such as:

1. Flagging a launchable activity as disabled in your manifest, then enabling it programmatically after the user has entered a license key, achieved some level or standing in a game, or any other criteria
2. Controlling whether a `BroadcastReceiver` registered in the manifest is hooked into the system or not, replicating the level of control you have with `registerReceiver()` while still taking advantage of the fact that a manifest-registered `BroadcastReceiver` can be started even if no other component of your application is running

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Searching with SearchManager

One of the firms behind the Open Handset Alliance — Google — has a teeny weeny Web search service, one you might have heard of in passing. Given that, it's not surprising that Android has some amount of built-in search capabilities.

Specifically, Android has "baked in" the notion of searching not only on the device for data, but over the air to Internet sources of data.

Your applications can participate in the search process, by triggering searches or perhaps by allowing your application's data to be searched.

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- content provider theory
- content provider implementations

Hunting Season

There are two types of search in Android: local and global. Local search searches within the current application; global search searches the Web via Google's search engine. You can initiate either type of search in a variety of ways, including:

1. You can call `onSearchRequested()` from a button or menu choice, which will initiate a local search (unless you override this method in your activity)
2. You can directly call `startSearch()` to initiate a local or global search, including optionally supplying a search string to use as a starting point

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

3. You can elect to have keyboard entry kick off a search via `setDefaultKeyMode()`, for either local search (`setDefaultKeyMode(DEFAULT_KEYS_SEARCH_LOCAL)`) or global search (`setDefaultKeyMode(DEFAULT_KEYS_SEARCH_GLOBAL)`)

In either case, the search appears as a set of UI components across the top of the screen, with a suggestion list (where available) and IME (where needed).

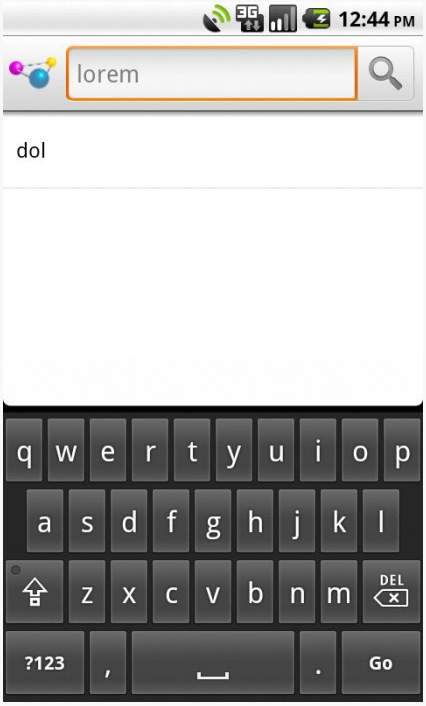


Figure 691: The Android local search popup, showing the IME and a previous search

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

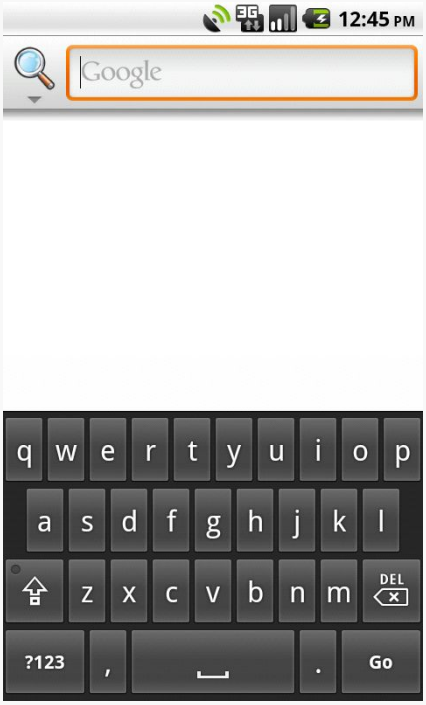


Figure 692: The Android global search popup

Where that search suggestion comes from for your local searches will be covered later in this chapter.

Search Yourself

Over the long haul, there will be two flavors of search available via the Android search system:

- Query-style search, where the user's search string is passed to an activity which is responsible for conducting the search and displaying the results
- Filter-style search, where the user's search string is passed to an activity on every keypress, and the activity is responsible for updating a displayed list of matches

Since the latter approach is decidedly under-documented, let's focus on the first one.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Craft the Search Activity

The first thing you are going to want to do if you want to support query-style search in your application is to create a search activity. While it might be possible to have a single activity be both opened from the launcher and opened from a search, that might prove somewhat confusing to users. Certainly, for the purposes of learning the techniques, having a separate activity is cleaner.

The search activity can have any look you want. In fact, other than watching for queries, a search activity looks, walks, and talks like any other activity in your system.

All the search activity needs to do differently is check the intents supplied to `onCreate()` (via `getIntent()`) and `onNewIntent()` to see if one is a search, and, if so, to do the search and display the results.

For example, let's look at the `Search/Lorem` sample application. This starts off as a version of the list-of-lorem-ipsum-words application seen in various places in this book. Now, we update it to support searching the list of words for ones containing the search string.

The main activity and the search activity both share a common layout: a `ListView` plus a `TextView` showing the selected entry:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent" >
  <TextView
    android:id="@+id/selection"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
  />
  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:drawSelectorOnTop="false"
  />
</LinearLayout>
```

In terms of Java code, most of the guts of the activities are poured into an abstract `LoremBase` class:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.search;

import android.app.ListActivity;
import android.app.SearchManager;
import android.content.Intent;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.ListAdapter;
import android.widget.ListView;
import android.widget.TextView;
import android.widget.Toast;
import java.util.ArrayList;
import org.xmlpull.v1.XmlPullParser;

abstract public class LoremBase extends ListActivity {
  abstract ListAdapter makeMeAnAdapter(Intent intent);

  private static final int LOCAL_SEARCH_ID = Menu.FIRST+1;
  private static final int GLOBAL_SEARCH_ID = Menu.FIRST+2;
  TextView selection;
  ArrayList<String> items=new ArrayList<String>();

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    selection=(TextView)findViewById(R.id.selection);

    try {
      XmlPullParser xpp=getResources().getXml(R.xml.words);

      while (xpp.getEventType()!=XmlPullParser.END_DOCUMENT) {
        if (xpp.getEventType()==XmlPullParser.START_TAG) {
          if (xpp.getName().equals("word")) {
            items.add(xpp.getAttributeValue(0));
          }
        }

        xpp.next();
      }
    }
    catch (Throwable t) {
      Toast
        .makeText(this, "Request failed: "+t.toString(), 4000)
        .show();
    }

    setDefaultKeyMode(DEFAULT_KEYS_SEARCH_LOCAL);

    onNewIntent(getIntent());
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void onNewIntent(Intent intent) {
    ListAdapter adapter=makeMeAnAdapter(intent);

    if (adapter==null) {
      finish();
    }
    else {
      setListAdapter(adapter);
    }
  }

  public void onListItemClick(ListView parent, View v, int position,
                long id) {
    selection.setText(parent.getAdapter().getItem(position).toString());
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    menu.add(Menu.NONE, LOCAL_SEARCH_ID, Menu.NONE, "Local Search")
            .setIcon(android.R.drawable.ic_search_category_default);
    menu.add(Menu.NONE, GLOBAL_SEARCH_ID, Menu.NONE, "Global Search")
            .setIcon(android.R.drawable.ic_menu_search)
            .setAlphabeticShortcut(SearchManager.MENU_KEY);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case LOCAL_SEARCH_ID:
        onSearchRequested();
        return(true);

      case GLOBAL_SEARCH_ID:
        startSearch(null, false, null, true);
        return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
}
```

This activity takes care of everything related to showing a list of words, even loading the words out of an XML resource. What it does not do is come up with the `ListAdapter` to put into the `ListView` – that is delegated to the subclasses.

The main activity — `LoremDemo` — just uses a `ListAdapter` for the whole word list:

```
package com.commonsware.android.search;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.content.Intent;
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;

public class LoremDemo extends LoremBase {
  @Override
  ListAdapter makeMeAnAdapter(Intent intent) {
    return(new ArrayAdapter<String>(this,
                      android.R.layout.simple_list_item_1,
                      items));
  }
}
```

The search activity, though, does things a bit differently.

First, it inspects the `Intent` supplied to the abstract `makeMeAnAdapter()` method. That `Intent` comes from either `onCreate()` or `onNewIntent()`. If the intent is an `ACTION_SEARCH`, then we know this is a search. We can get the search query and, in the case of this silly demo, spin through the loaded list of words and find only those containing the search string. That list then gets wrapped in a `ListAdapter` and returned for display:

```
  @Override
  ListAdapter makeMeAnAdapter(Intent intent) {
    ListAdapter adapter=null;

    if (intent.getAction().equals(Intent.ACTION_SEARCH)) {
      String query=intent.getStringExtra(SearchManager.QUERY);
      List<String> results=searchItems(query);

      adapter=new ArrayAdapter<String>(this,
                        android.R.layout.simple_list_item_1,
                        results);
      setTitle("LoremSearch for: "+query);
    }

    return(adapter);
  }
```

The logic in the `searchItems()` method that actually finds the matches looks like:

```
    List<String> results=new ArrayList<String>();

    for (String item : items) {
      if (item.indexOf(query)>-1) {
        results.add(item);
      }
    }

    return(results);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We will see the rest of that method later in this chapter.

Update the Manifest

While this implements search, it doesn't tie it into the Android search system. That requires a few changes to the auto-generated `AndroidManifest.xml` file:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.search">

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <supports-screens
    android:largeScreens="false"
    android:normalScreens="true"
    android:smallScreens="false"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="Lorem Ipsum">
    <activity
      android:name=".LoremDemo"
      android:label="LoremDemo">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>

      <meta-data
        android:name="android.app.default_searchable"
        android:value=".LoremSearch"/>
    </activity>
    <activity
      android:name=".LoremSearch"
      android:label="LoremSearch"
      android:launchMode="singleTop">
      <intent-filter>
        <action android:name="android.intent.action.SEARCH"/>

        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>

      <meta-data
        android:name="android.app.searchable"
        android:resource="@xml/searchable"/>
    </activity>
  </application>

</manifest>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The changes that are needed are:

- The `LoremDemo` main activity gets a `meta-data` element, with an `android:name` of `android.app.default_searchable` and a `android:value` of the search implementation class (`.LoremSearch`)
- The `LoremSearch` activity gets an intent filter for `android.intent.action.SEARCH`, so search intents will be picked up
- The `LoremSearch` activity is set to have `android:launchMode = "singleTop"`, which means at most one instance of this activity will be open at any time, so we don't wind up with a whole bunch of little search activities cluttering up the activity stack
- Add `android:label` and `android:icon` attributes to the `application` element — these will influence how your application appears in the Quick Search Box among other places
- The `LoremSearch` activity gets a `meta-data` element, with an `android:name` of `android.app.searchable` and an `android:value` of an XML resource containing more information about the search facility offered by this activity (`@xml/searchable`)

```
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
  android:label="@string/searchLabel"
  android:hint="@string/searchHint"

android:searchSuggestAuthority="com.commonsware.android.search.LoremSuggestionProvider"
  android:searchSuggestSelection=" ? "
  android:searchSettingsDescription="@string/global"
  android:includeInGlobalSearch="true"
/>
```

That XML resource provides many bits of information, of which only two are needed for simple search-enabled applications:

- What name should appear in the search domain button to the left of the search field, identifying to the user where she is searching (`android:label`)
- What hint text should appear in the search field, to give the user a clue as to what they should be typing in (`android:hint`)

The other attributes found in that file, and the other search-related bits found in the manifest, will be covered later in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Searching for Meaning In Randomness

Given all that, search is now available — Android knows your application is searchable, what search domain to use when searching from the main activity, and the activity knows how to do the search.

The options menu for this application has both local and global search options. In the case of local search, we just call `onSearchRequested()`; in the case of global search, we call `startSearch()` with true in the last parameter, indicating the scope is global.

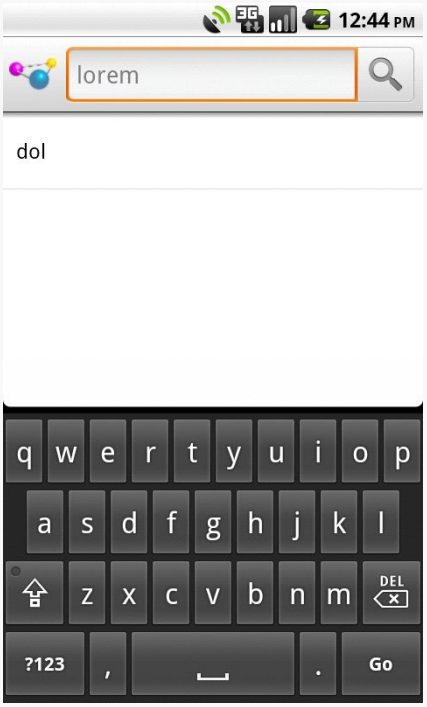


Figure 693: The Lorem sample application, showing the local search popup

Typing in a letter or two, then clicking Search, will bring up the search activity and the subset of words containing what you typed, with your search query in the activity title bar:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

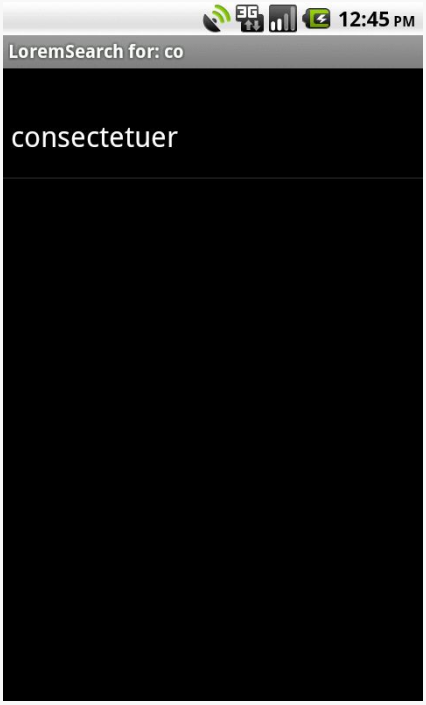


Figure 694: The results of searching for 'co' in the Lorem search sample

You can get the same effect if you just start typing in the main activity, since it is set up for triggering a local search.

May I Make a Suggestion?

When you do a global search, you are given "suggestions" of search words or phrases that may be what you are searching for, to save you some typing on a small keyboard:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

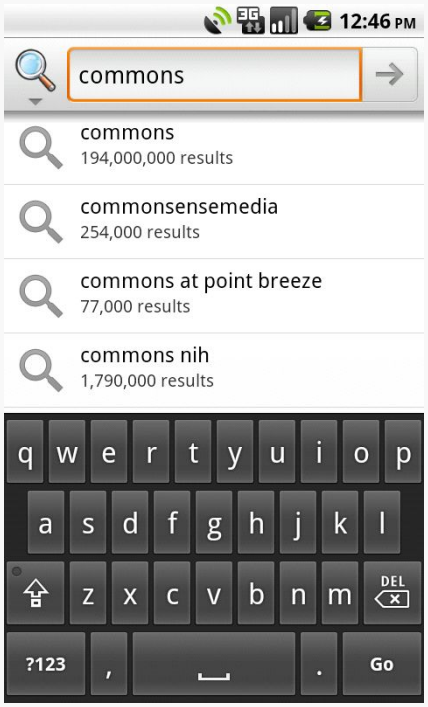


Figure 695: Search suggestions after typing some letters in global search

Your application, if it chooses, can offer similar suggestions. Not only will this give you the same sort of drop-down effect as you see with the global search above, but it also ties neatly into the Quick Search Box, as we will see later in this chapter.

To provide suggestions, you need to implement a `ContentProvider` and tie that provider into the search framework. You have two major choices for implementing a suggestion provider: use the built-in "recent" suggestion provider, or create your own from scratch.

SearchRecentSuggestionsProvider

The "recent" suggestions provider gives you a quick and easy way to remember past searches and offer those as suggestions on future searches.

To use this facility, you must first create a custom subclass of `SearchRecentSuggestionsProvider`. Your subclass may be very simple, perhaps just a two-line constructor with no other methods. However, since Android does not automatically record recent queries for you, you will also need to give your search activity a way to record them such that the recent-suggestions provider can offer them as suggestions in the future.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Below, we have a `LoremSuggestionProvider`, extending `SearchRecentSuggestionsProvider`, that also supplies a "bridge" for the search activity to record searches:

```
package com.commonsware.android.search;

import android.content.Context;
import android.content.SearchRecentSuggestionsProvider;
import android.provider.SearchRecentSuggestions;

public class LoremSuggestionProvider
  extends SearchRecentSuggestionsProvider {
  private static String
AUTH="com.commonsware.android.search.LoremSuggestionProvider";

  static SearchRecentSuggestions getBridge(Context ctxt) {
    return(new SearchRecentSuggestions(ctxt, AUTH,
                                       DATABASE_MODE_QUERIES));
  }

  public LoremSuggestionProvider() {
      super();

      setupSuggestions(AUTH, DATABASE_MODE_QUERIES);
  }
}
```

The constructor, besides the obligatory chain to the superclass, simply calls `setupSuggestions()`. This takes two parameters:

1. The authority under which you will register this provider in the manifest (see below)
2. A flag indicating where the suggestions will come from — in this case, we supply the required `DATABASE_MODE_QUERIES` flag

Of course, since this is a `ContentProvider`, you will need to add it to your manifest:

```
    <provider
      android:name=".LoremSuggestionProvider"

android:authorities="com.commonsware.android.search.LoremSuggestionProvider"/>
```

The other thing that `LoremSuggestionProvider` has is a static method that creates a properly-configured instance of a `SearchRecentSuggestions` object. This object knows how to save search queries to the database that the content provider uses, so they will be served up as future suggestions. It needs to know the same authority and flag that you provide to `setupSuggestions()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

That SearchRecentSuggestions is then used by our LoremSearch class, inside its searchItems() method that actually examines the list of Latin words for matches:

```
private List<String> searchItems(String query) {
  LoremSuggestionProvider
    .getBridge(this)
    .saveRecentQuery(query, null);

  List<String> results=new ArrayList<String>();

  for (String item : items) {
    if (item.indexOf(query)>-1) {
      results.add(item);
    }
  }

  return(results);
}
```

In this case, we always record the search, though you can imagine that some applications might not save searches that are invalid for one reason or another.

Custom Suggestion Providers

If you want to provide search suggestions based on something else – actual data, searches conducted by others that you aggregate via a Web service, etc. — you will need to implement your own ContentProvider that supplies that information. As with SearchRecentSuggestionsProvider, you will need to add your ContentProvider to the manifest so that Android knows it exists.

The details for doing this will be covered in a future edition of this book. For now, you are best served with the Android SearchManager documentation on the topic.

Integrating Suggestion Providers

Before your suggestions will appear, though, you need to tell Android to use your ContentProvider as the source of suggestions. There are two attributes on your searchable XML that make this connection:

1. android:searchSuggestAuthority indicates the content authority for your suggestions — this is the same authority you used for your ContentProvider
2. android:searchSuggestSelection is how the suggestion should be packaged as a query in the ACTION_SEARCH Intent — unless you have some reason to do otherwise, " ? " is probably a fine value to use

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The result is that when we do our local search, we get the drop-down of past searches as suggestions:

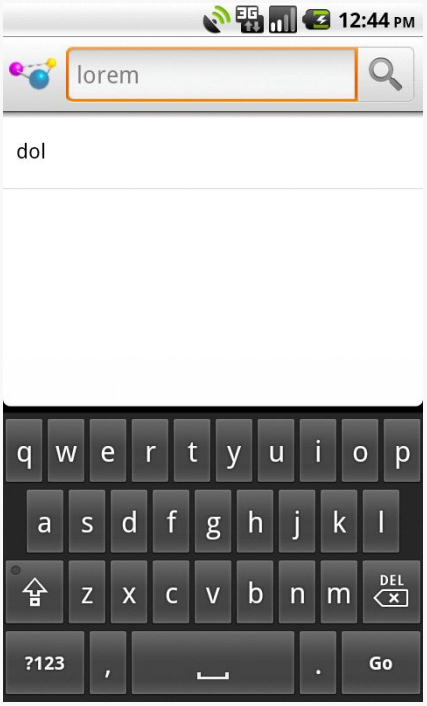


Figure 696: The Android local search popup, showing the IME and a previous search

There is also a `clearHistory()` method on `SearchRecentSuggestions` that you can use, perhaps from a menu choice, to clear out the search history, in case it is cluttered beyond usefulness.

Putting Yourself (Almost) On Par with Google

The Quick Search Box is Android's new term for the search widget at the top of the home screen. This is the same UI that appears when your application starts a global search. When you start typing, it shows possible matches culled from both the device and the Internet. If you choose one of the suggestions, it takes you to that item – choose a contact, and you visit the contact in the Contacts application. If you choose a Web search term, or you just submit whatever you typed in, Android will fire up a Browser instance showing you search results from Google. The order of suggestions is adaptive, as Android will attempt to show the user the sorts of things the user typically searches for (e.g., if the user clicks on contacts a lot in prior searches, it may prioritize suggested contacts in the suggestion list).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Your application can be tied into the Quick Search Box. However, it is important to understand that being in the Quick Search Box does *not* mean that your content will be searched. Instead, your *suggestions provider* will be queried based on what the user has typed in, and those suggestions will be blended into the overall results.

And, your application will not show up in Quick Search Box suggestions automatically — the user has to "opt in" to have your results included.

And, until the user demonstrates an interest in your results, your application's suggestions will be buried at the bottom of the list.

This means that integrating with the Quick Search Box, while still perhaps valuable, is not exactly what some developers will necessarily have in mind. That being said, here is how to achieve this integration.

NOTE: there is some flaw in the Android 2.2 emulator that prevents this from working, though it works fine on Android 2.2 hardware.

Implement a Suggestions Provider

Your first step is to implement a suggestions provider, as described in the previous section. Again, Android does not search your application, but rather queries your suggestions provider. If you do not have a suggestions provider, you will not be part of the Quick Search Box. As we will see below, this approach means you will need to think about what sort of suggestion provider to create.

Augment the Metadata

Next, you need to tell Android to tie your application into the Quick Search Box suggestion list. To do that, you need to add the `android:includeInGlobalSearch` attribute to your `searchable` XML, setting it to `true`. You probably also should consider adding the `android:searchSettingsDescription`, as this will be shown in the UI for the user to configure what suggestions the Quick Search Box shows.

Convince the User

Next, the user needs to activate your application to be included in the Quick Search Box suggestion roster. To do that, the user needs to go into Settings, find where the "Searchable Items" screen is, and check the checkbox associated with your application:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

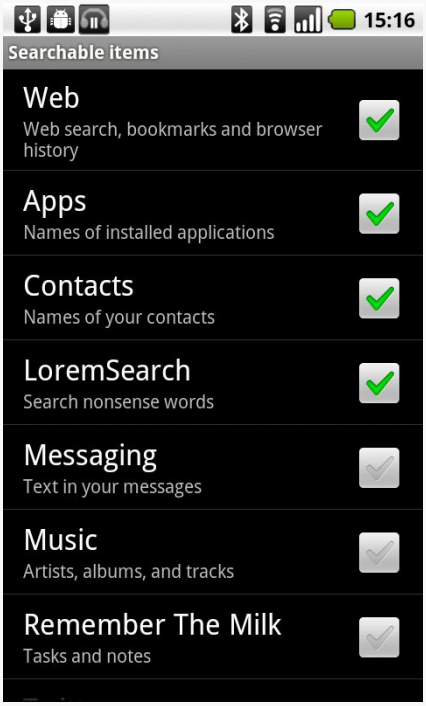


Figure 697: The Searchable Items settings screen

The precise location of the "Searchable Items" screen varies by OS version and possibly by device.

Your application's label and the value of `android:searchSettingsDescription` are what appears to the left of the checkbox.

You have no way of toggling this on yourself — the user has to do it. You may wish to mention this in the documentation for your application.

The Results

If you and the user do all of the above, now when the user initiates a search, your suggestions will be poured into the suggestions list, at the bottom:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

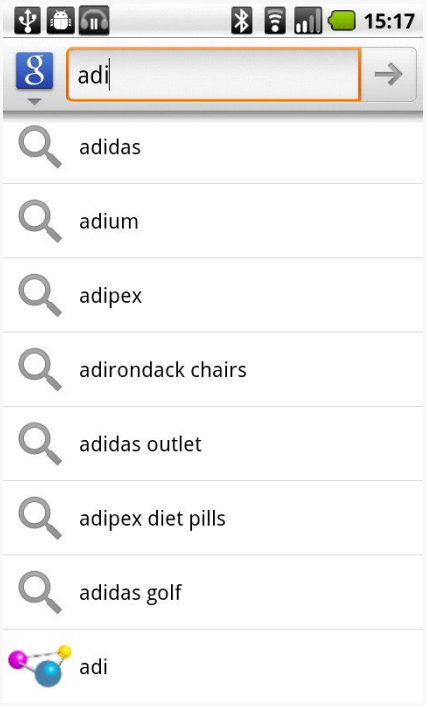


Figure 698: The Quick Search Box, showing application-supplied suggestions

On versions of Android prior to 2.2, to actually see your suggestions, the user also needs to click the arrow to "fold open" the actual suggestions:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

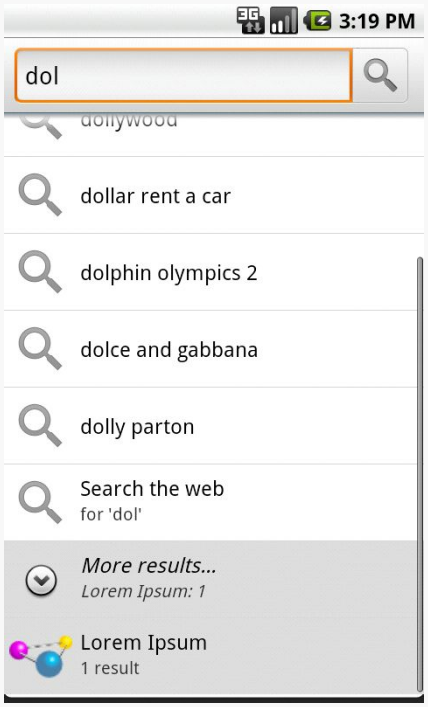


Figure 699: The Quick Search Box, showing another placeholder for application-supplied suggestions

Even here, we do not see the actual suggestion. However, if the user clicks on that item, your suggestions then take over the list:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

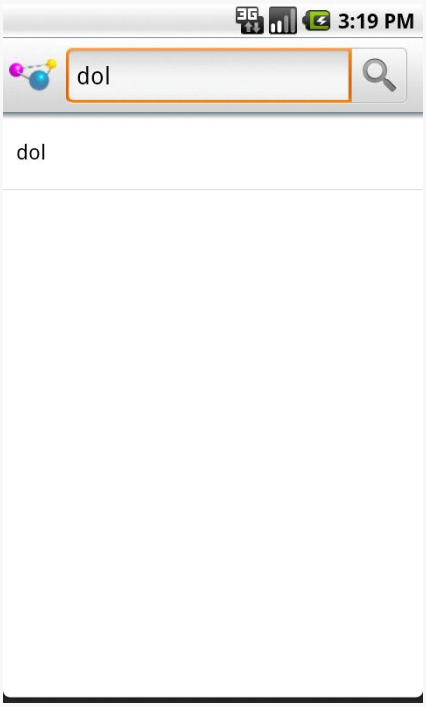


Figure 700: The Quick Search Box, showing application-supplied suggestions

Again, Android is not showing *actual data* from your application – our list of Latin words does not contain the value "dol". Instead, Android is showing *suggestions* from your suggestion provider based on what the user typed in. In this case, our application's suggestion provider is based on the built-in `SearchRecentSuggestionsProvider` class, meaning the suggestions are *past queries*, not actual results.

Hence, what you want to have appear in the Quick Search Box suggestion list will heavily influence what sort of suggestion provider you wish to create. While a `SearchRecentSuggestionsProvider` is simple, what you get in the Quick Search Box suggestions may not be that useful to users. Instead, you may wish to create your own custom suggestions provider, providing suggestions from actual data or other more useful sources, perhaps in addition to saved searches.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Remote Services and the Binding Pattern

Earlier in this book, we covered using services by sending commands to them to be processed. That "command pattern" is one of two primary means of interacting with a service — the binding pattern is the other. With the binding pattern, your service exposes a more traditional API, in the form of a "binder" object with methods of your choosing. On the plus side, you get a richer interface. However, it more tightly ties your activity to your service, which may cause you problems with configuration changes.

Either the command pattern or the binding pattern can be used, if desired, across process boundaries, with the client being some third-party application. In either case, you will need to export your service via an `<intent-filter>`. And, in the case of the binding pattern, your "binder" implementation will have some restrictions.

This chapter covers the binding pattern for local services, plus inter-process commands and binding (a.k.a., remote services).

Prerequisites

Understanding this chapter requires that you have read the chapters on:

- broadcast Intents
- service theory

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Binding Pattern

Implementing the binding pattern requires work on both the service side and the client side. The service will need to have a full implementation of the `onBind()` method, which typically just returns `null` or throws some sort of runtime exception for a service solely implementing the command pattern. And, the client (e.g., an activity) will need to ask to bind to the service, instead of (or perhaps in addition to) starting the service.

What the Service Does

The service implements a subclass of `Binder` that represents the service's exposed API. For a local service, your `Binder` can have pretty much whatever methods you want: method names, parameters, return types, and exceptions thrown are up to you. When you get into remote services, your `Binder` implementation will be substantially more constrained, to support inter-process communication.

Then, your `onBind()` method returns an instance of the `Binder`.

What the Client Does

Clients call `bindService()`, supplying the `Intent` that identifies the service, a `ServiceConnection` object representing the client side of the binding, and an optional `BIND_AUTO_CREATE` flag. As with `startService()`, `bindService()` is asynchronous. The client will not know anything about the status of the binding until the `ServiceConnection` object is called with `onServiceConnected()`. This not only indicates the binding has been established, but for local services it provides the Binder object that the service returned via `onBind()`. At this point, the client can use the `Binder` to ask the service to do work on its behalf.

Note that if the service is not already running, and if you provide `BIND_AUTO_CREATE`, then the service will be created first before being bound to the client. If you skip `BIND_AUTO_CREATE`, and the service is not already running, `bindService()` is supposed to return `false`, indicating there was no existing service to bind to. However, in actuality, Android returns `true`, due to an apparent bug.

Eventually, the client will need to call `unbindService()`, to indicate it no longer needs to communicate with the service. For example, an activity might call `bindService()` in its `onCreate()` method, then call `unbindService()` in its `onDestroy()` method. Once you call `unbindService()`, your `Binder` object is no

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

longer safe to be used by the client. If there are no other bound clients to the service, Android will shut down the service as well, releasing its memory. Hence, we do not need to call `stopService()` ourselves – Android handles that, if needed, as a side effect of unbinding.

Your `ServiceConnection` object will also need an `onServiceDisconnected()` method. This will be called only if there is an unexpected disconnection, such as the service crashing with an unhandled exception.

A Binding Sample

In the chapter introducing services, we saw a sample app that would download a file off of a Web server. That sample used the command pattern, telling the service what to download via an `Intent` extra. In this chapter, we will review a few variations of that sample, all of which use the binding pattern instead of the command pattern.

Right now, we are focused on local services, and so the `Binding/Local` sample project does the download via a local bound service.

We start by defining an interface that will serve as the "contract" between the client (fragment) and service. This interface, `IDownload`, contains a single `download()` method:

```
package com.commonsware.android.advservice.binding;

// Declare the interface.
interface IDownload {
  void download(String url);
}
```

Our service, `DownloadService`, implements just one method, `onBind()`, which returns an instance of a `DownloadBinder`:

```
package com.commonsware.android.advservice.binding;

import android.app.Service;
import android.content.Intent;
import android.net.Uri;
import android.os.Binder;
import android.os.Environment;
import android.os.IBinder;
import android.util.Log;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class DownloadService extends Service {
  @Override
  public IBinder onBind(Intent intent) {
    return(new DownloadBinder());
  }

  private static class DownloadBinder extends Binder implements IDownload {
    @Override
    public void download(String url) {
      new DownloadThread(url).start();
    }
  }

  private static class DownloadThread extends Thread {
    String url=null;

    DownloadThread(String url) {
      this.url=url;
    }

    @Override
    public void run() {
      try {
        File root=

Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);

        root.mkdirs();

        File output=new File(root, Uri.parse(url).getLastPathSegment());

        if (output.exists()) {
          output.delete();
        }

        HttpURLConnection c=(HttpURLConnection)new URL(url).openConnection();

        FileOutputStream fos=new FileOutputStream(output.getPath());
        BufferedOutputStream out=new BufferedOutputStream(fos);

        try {
          InputStream in=c.getInputStream();
          byte[] buffer=new byte[8192];
          int len=0;

          while ((len=in.read(buffer)) >= 0) {
            out.write(buffer, 0, len);
          }

          out.flush();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        }
        finally {
          fos.getFD().sync();
          out.close();
          c.disconnect();
        }
      }
      catch (IOException e2) {
        Log.e("DownloadJob", "Exception in download", e2);
      }
    }
  }
}
```

`DownloadBinder` implements the `IDownload` interface. Its `download()` method, in turn, forks a `DownloadThread` to perform the download in the background — remember, for local services, the methods you invoke on the `Binder` are executed on whatever thread you call them on.

Our fragment, `DownloadFragment`, loads our layout, `res/layout/main.xml`, containing a `Button` to trigger the download:

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/go"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
    android:text="@string/go"/>
```

The implementation of `onCreateView()` simply loads that layout, gets the `Button`, sets up the fragment as being the click listener for the `Button`, and disables the `Button`:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.main, container, false);

    btn=(Button)result.findViewById(R.id.go);
    btn.setOnClickListener(this);
    btn.setEnabled(binding!=null);

    return(result);
  }
```

The reason why we disable the `Button` is because we are not connected to our service at this point, and until we are, we cannot allow the user to try to download a file.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In `onAttach()` of our fragment, we bind to the service:

```
@Override
public void onAttach(Activity host) {
  super.onAttach(host);

  appContext=(Application)host.getApplicationContext();
  appContext.bindService(new Intent(getActivity(),
                                    DownloadService.class), this,
                         Context.BIND_AUTO_CREATE);
}
```

You will notice something curious here: `getApplicationContext()`. Technically, we could bind to the service directly from the `Activity`, by calling `bindService()` on it, as `bindService()` is a method on `Context`. However, our service binding represents some state, and it is possible that this state will hold a reference to the `Context` that created the binding. In that case, we run the risk of leaking our original activity during a configuration change. The `getApplicationContext()` method returns the global `Application` singleton, which is a `Context` suitable for binding, but one that cannot be leaked, since it is already in a global scope. In effect, it is "pre-leaked".

Some time after `onAttach()` is called and we call `bindService()`, our `onServiceConnected()` method will be called, as we designated our fragment to be the `ServiceConnection`. Here, we can cast the `IBinder` object we receive to be our `IDownload` interface to the service, and we can enable the `Button`:

```
@Override
public void onServiceConnected(ComponentName className, IBinder binder) {
  binding=(IDownload)binder;
  btn.setEnabled(true);
}
```

Since we are implementing the `ServiceConnection` interface, our fragment also needs to implement the `onServiceDisconnected()` method, invoked if our service crashes. Here, we delegate responsibility to a `disconnect()` private method, which removes our link to the `IDownload` object and disables our `Button`:

```
@Override
public void onServiceDisconnected(ComponentName className) {
  disconnect();
}

private void disconnect() {
  binding=null;
  btn.setEnabled(false);
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

And, when our fragment is destroyed, we unbind from the service (using the same `Context` as before, from `getApplicationContext()`) and `disconnect()`:

```
  @Override
  public void onDestroy() {
    appContext.unbindService(this);
    disconnect();

    super.onDestroy();
  }
```

However, in between `onServiceConnected()` and either `onServiceDisconnected()` or `onDestroy()`, the user can click the `Button`, which will trigger the download via a call to `download()` on our `IDownload` instance:

```
  @Override
  public void onClick(View view) {
    binding.download(TO_DOWNLOAD);
  }
```

The `DownloadBindingDemo` activity adds our `DownloadFragment` via a `FragmentTransaction`:

```
package com.commonsware.android.advservice.binding;

import android.app.Activity;
import android.os.Bundle;

public class DownloadBindingDemo extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getFragmentManager().findFragmentById(android.R.id.content) == null) {
      getFragmentManager().beginTransaction()
                          .add(android.R.id.content,
                               new DownloadFragment()).commit();
    }
  }
}
```

The fact that we are using a `FragmentTransaction` means that our `DownloadFragment` can be retained, which we set up in `onCreate()` of the fragment:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    setRetainInstance(true);
  }
```

This allows the fragment — serving as our `ServiceConnection` — to survive a configuration change, so we can cleanly unbind from the service when `onDestroy()` is called.

Starting and Binding

Some developers will use *both* `startService()` and `bindService()` at the same time. The typical argument is that they need frequent updates from the service (e.g., percentage of progress, for updating a `ProgressBar`) in the client and are concerned about the overhead of sending broadcasts.

With the advent of `LocalBroadcastManager` and other event bus implementations, binding to a service you are using with `startService()` should no longer be necessary.

When IPC Attacks!

If you wish to extend the binding pattern to serve in the role of IPC, whereby other processes can get at your `Binder` and call its methods, you will need to use AIDL: the Android Interface Description Language. If you have used IPC mechanisms like SOAP, XML-RPC, DCOM, CORBA, or the like, you will recognize the notion of IDL. AIDL describes the public IPC interface, and Android supplies tools to build the client and server side of that interface.

With that in mind, let's take a look at AIDL and IPC.

Write the AIDL

IDLs are frequently written in a "language-neutral" syntax. AIDL, on the other hand, looks a lot like a Java interface file. For example, here is some AIDL:

```
package com.commonsware.android.advservice.remotebinding;

// Declare the interface.
interface IDownload {
  void download(String url);
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

As you will notice, this looks suspiciously like the regular Java interface we used in the simple binding example earlier in this chapter.

As with a Java interface, you declare a package at the top. As with a Java interface, the methods are wrapped in an interface declaration (`interface IDownload { ... }`). And, as with a Java interface, you list the methods you are making available.

The differences, though, are critical.

First, not every Java type can be used as a parameter. Your choices are:

1. Primitive values (`int`, `float`, `double`, `boolean`, etc.)
2. `String` and `CharSequence`
3. `List` and `Map` (from `java.util`)
4. Any other AIDL-defined interfaces
5. Any Java classes that implement the `Parcelable` or `Serializable` interface

In the case of the latter two categories, you need to include `import` statements referencing the names of the classes or interfaces that you are using (e.g., `import com.commonsware.android.ISomething`). This is true even if these classes are in your own package — you have to import them anyway.

Next, parameters can be classified as `in`, `out`, or `inout`. Values that are `out` or `inout` can be changed by the service and those changes will be propagated back to the client. Primitives (e.g., `int`) can only be `in`.

Also, you cannot throw any exceptions. You will need to catch all exceptions in your code, deal with them, and return failure indications some other way (e.g., error code return values).

Name your AIDL files with the `.aidl` extension and place them in the proper directory based on the package name:

- For native Android Studio projects, this will be an `aidl/` directory in your `src/` sourceset, as a peer of your `java/` directory, with the same sort of subdirectories-based-on-the-Java-package approach as you use for regular Java source code
- For Eclipse-compatible projects, the `.aidl` files will go alongside your `.java` files in the `src/` directory tree

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

When you build your project, either via an IDE or via Ant, the `aidl` utility from the Android SDK will translate your AIDL into a server stub and a client proxy.

Implement the Interface

Given the AIDL-created server stub, now you need to implement the service, either directly in the stub, or by routing the stub implementation to other methods you have already written.

The mechanics of this are fairly straightforward:

1. Create a private instance of the AIDL-generated `.Stub` class (e.g., `IDownload.Stub`)
2. Implement methods matching up with each of the methods you placed in the AIDL
3. Return this private instance from your `onBind()` method in the `Service` subclass

Note that AIDL IPC calls are synchronous, and so the caller is blocked until the IPC method returns. Hence, your services need to be quick about their work.

We will see examples of service stubs later in this chapter.

Service From Afar

So, given our AIDL description, let us examine a sample implementation, using AIDL for a remote service.

Our sample applications — shown in the `Binding/Remote/Service` and `Binding/Remote/Client` sample projects — simply move the service logic into a separate project from the client logic.

Service Names

To bind to a service's AIDL-defined API, you need to craft an `Intent` that can identify the service in question. In the case of a local service, that Intent can use the local approach of directly referencing the service class.

Obviously, that is not possible in a remote service case, where the service class is not in the same process, and may not even be known by name to the client.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

When you define a service to be used by remote, you need to add an `<intent-filter>` element to your service declaration in the manifest, indicating how you want that service to be referred to by clients. The manifest for `RemoteService` is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.advservice.remotebinding.svc"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk
    android:minSdkVersion="14"
    android:targetSdkVersion="14"/>

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
        android:theme="@android:style/Theme.Holo.Light.DarkActionBar">
        <service android:name=".DownloadService">
            <intent-filter>
                <action
android:name="com.commonsware.android.advservice.remotebinding.IDownload" />
            </intent-filter>
        </service>
  </application>

</manifest>
```

Here, we say that the service can be identified by the name `com.commonsware.android.advservice.IDownload`. So long as the client uses this name to identify the service, it can bind to that service's API.

In this case, the name is not an implementation, but the AIDL API, as you will see below. In effect, this means that so long as some service exists on the device that implements this API, the client will be able to bind to something.

Remote Services and Implicit Intents

We are used to a device having multiple activities that can respond to the same `<intent-filter>`. In that case, by default, the user will see a chooser if we try to start one of those activities.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

We are used to a device having multiple `BroadcastReceiver` components that can respond to the same `<intent-filter>` (or `IntentFilter`). In that case, in a regular broadcast, all eligible receivers will receive it.

We are used to it being impossible to have multiple `ContentProvider` components with the same authority, as the second one fails on install with an `INSTALL_FAILED_CONFLICTING_PROVIDER` error.

What happens if there are *two* (or more) services installed on the device that claim to support the same `<intent-filter>`, but have different package names? You might think that this would fail on install, as happens with providers with duplicate authorities. Alas, it does not... prior to Android 5.0. Instead, the higher-priority `<intent-filter>` gets it (set via the `android:priority` attribute). If 2+ implementations have the same priority, the first one installed wins.

So, if we have `BadService` and `GoodService`, both responding to the same `<intent-filter>`, and a client app tries to communicate to `GoodService` via the implicit `Intent` matching that `<intent-filter>`, it might *actually* be communicating with `BadService`, simply because `BadService` was installed first. The user is oblivious to this.

Android 5.0 solves this by preventing binding using an implicit `Intent`. This, however, presents a conundrum:

- We cannot bind using an implicit `Intent`
- We do not know how to construct an explicit `Intent` identifying the desired service, as that might be from a third-party app

As you will see, when we examine the client side of this sample, we have to use `PackageManager` to convert an implicit `Intent` into a valid explicit `Intent` for our service. This not only allows us to comply with the Android 5.0 binding restriction, but it gives us an opportunity to detect and handle the cases where there is no matching service (e.g., the service app has not yet been installed) or when there is more than one matching service (e.g., `BadService` and `GoodService`). And the techniques that all of this uses works on pretty much any version of Android, so while we *need* them for Android 5.0 and higher, we can *use* them anywhere.

The Service

Beyond the manifest, the service implementation is not too unusual. There is the AIDL interface, `IDownload`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
package com.commonsware.android.advservice.remotebinding;

// Declare the interface.
interface IDownload {
  void download(String url);
}
```

And there is the actual service class itself, `DownloadService`:

```
package com.commonsware.android.advservice.remotebinding.svc;

import android.app.Service;
import android.content.Intent;
import android.net.Uri;
import android.os.Environment;
import android.os.IBinder;
import android.util.Log;
import com.commonsware.android.advservice.remotebinding.IDownload;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class DownloadService extends Service {
  @Override
  public IBinder onBind(Intent intent) {
    return(new DownloadBinder());
  }

  private static class DownloadBinder extends IDownload.Stub {
    @Override
    public void download(String url) {
      new DownloadThread(url).start();
    }
  }

  private static class DownloadThread extends Thread {
    String url=null;

    DownloadThread(String url) {
      this.url=url;
    }

    @Override
    public void run() {
      try {
        File root=

Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        root.mkdirs();

        File output=new File(root, Uri.parse(url).getLastPathSegment());

        if (output.exists()) {
          output.delete();
        }

        HttpURLConnection c=(HttpURLConnection)new URL(url).openConnection();

        FileOutputStream fos=new FileOutputStream(output.getPath());
        BufferedOutputStream out=new BufferedOutputStream(fos);

        try {
          InputStream in=c.getInputStream();
          byte[] buffer=new byte[8192];
          int len=0;

          while ((len=in.read(buffer)) >= 0) {
            out.write(buffer, 0, len);
          }

          out.flush();
        }
        finally {
          fos.getFD().sync();
          out.close();
          c.disconnect();
        }
      }
      catch (IOException e2) {
        Log.e("DownloadJob", "Exception in download", e2);
      }
    }
  }
}
```

This is identical to the local binding example, with one key difference: `DownloadService` now extends `IDownload.Stub` rather than the generic `Binder` class.

The Client

The client — a revised version of `DownloadFragment` — connects to the remote service to ask it to download the file on the user's behalf. This has three changes of note over our original local implementation.

First, when we call `download()` on the `IDownload` object, we need to catch a `RemoteException`. This will thrown if the service crashes during our request or otherwise is unable to return properly:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
@Override
public void onClick(View view) {
  try {
    binding.download(TO_DOWNLOAD);
  }
  catch (RemoteException e) {
    Log.e(getClass().getSimpleName(), "Exception requesting download", e);
    Toast.makeText(getActivity(), e.getMessage(), Toast.LENGTH_LONG).show();
  }
}
```

Second, our `onServiceConnected()` uses `IDownload.Stub.asInterface()` to convert the raw `IBinder` into an `IDownload` object for use:

```
@Override
public void onServiceConnected(ComponentName className, IBinder binder) {
  binding=IDownload.Stub.asInterface(binder);
  btn.setEnabled(true);
}
```

Third, our binding logic in `onAttach()` is significantly more complicated:

```
@Override
public void onAttach(Activity host) {
  super.onAttach(host);

  appContext=(Application)host.getApplicationContext();

  Intent implicit=new Intent(IDownload.class.getName());
  List<ResolveInfo> matches=host.getPackageManager()
                                .queryIntentServices(implicit, 0);

  if (matches.size()==0) {
    Toast.makeText(host, "Cannot find a matching service!",
                   Toast.LENGTH_LONG).show();
  }
  else if (matches.size()>1) {
    Toast.makeText(host, "Found multiple matching services!",
                   Toast.LENGTH_LONG).show();
  }
  else {
    Intent explicit=new Intent(implicit);
    ServiceInfo svcInfo=matches.get(0).serviceInfo;
    ComponentName cn=new ComponentName(svcInfo.applicationInfo.packageName,
                                       svcInfo.name);

    explicit.setComponent(cn);
    appContext.bindService(explicit, this, Context.BIND_AUTO_CREATE);
  }
}
```

Here, we:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Get the `Application` singleton `Context` as before
- Craft an implicit `Intent` for the service, using the appropriate action string (which, in this case, happens to be the fully-qualified name of the `IDownload` interface)
- Use `PackageManager` and `queryIntentServices()` to find out all services that implement a matching `<intent-filter>` for that implicit `Intent`
- Fail with a `Toast` if there is not exactly one such service
- Use the `ServiceInfo` object from our `queryIntentServices()` call to craft an explicit `Intent`, with the same structure as the implicit `Intent` had, but also with the actual matched component (via `setComponent()`)
- Use the explicit `Intent` to bind to the service

Note that the client needs its own copy of `IDownload.aidl`. After all, it is a totally separate application, and therefore does not share source code with the service. In a production environment, we might craft and distribute a JAR file that contains the `IDownload` classes, so both client and service can work off the same definition (see the upcoming chapter on reusable components). For now, we will just have a copy of the AIDL.

If you compile both applications and upload them to the device, then start up the client, you can have the service download the file.

Servicing the Service

However, we do not get any result back from the service to know if the download succeeded or failed. That is likely to be rather important information for the user.

In principle, `download()` could return some success-or-failure indication... but then we would have a blocking call. Neither the client nor the service could proceed until the download is completed. That would require the client to manage its own background thread, which is a minor hassle. It also means that the service ties up one of a limited number of "`Binder` threads", which is not a good idea.

Another approach would be to pass some sort of callback object with `download()`, such that the server could run the script asynchronously and invoke the callback on success or failure. This, though, implies that there is some way to have the client export an API to the service.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Fortunately, this is eminently doable, as you will see in this section, and the accompanying samples (Binding/Callback/Service and Binding/Callback/Client).

Callbacks via AIDL

AIDL does not have any concept of direction. It just knows interfaces, proxies, and stub implementations. In the preceding example, we used AIDL to have the service flesh out the stub implementation and have the client access the service via the AIDL-defined interface. However, there is nothing magic about services implementing interfaces and clients accessing them — it is equally possible to reverse matters and have the client implement something the service uses via an interface.

So, for example, we could create an `IDownloadCallback.aidl` file:

```
package com.commonsware.android.advservice.callbackbinding;

// Declare the interface.
interface IDownloadCallback {
  void onSuccess();
  void onFailure(String msg);
}
```

Then, we can augment `IDownload` itself, to pass an `IDownloadCallback` with `download()`:

```
package com.commonsware.android.advservice.callbackbinding;

import com.commonsware.android.advservice.callbackbinding.IDownloadCallback;

// Declare the interface.
interface IDownload {
  void download(String url, IDownloadCallback cb);
}
```

Notice that we need to specifically import `IDownloadCallback`, just like we might import some "regular" Java interface. And, as before, we need to make sure the client and the server are working off of the same AIDL definitions, so these two AIDL files need to be replicated across each project.

But other than that one little twist, this is all that is required, at the AIDL level, to have the client pass a callback object to the service: define the AIDL for the callback and add it as a parameter to some service API call.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Of course, there is a little more work to do on the client and server side to make use of this callback object.

Revising the Client

On the client, we need to implement an `IDownloadCallback`. In `onSuccess()` and `onFailure()` we can do something like raise a `Toast`.

The catch is that we cannot be certain we are being called on the UI thread in our callback object. In fact, it is almost assured that we are not. So, we need to get our work moved over to the main application thread. To do that, this sample uses greenrobot's EventBus, having the `IDownloadCallback.Stub` post a `CallbackEvent` indicating success or failure:

```
  IDownloadCallback.Stub cb=new IDownloadCallback.Stub() {
    @Override
    public void onSuccess() throws RemoteException {
      EventBus.getDefault().post(new CallbackEvent(true, null));
    }

    @Override
    public void onFailure(String msg) throws RemoteException {
      EventBus.getDefault().post(new CallbackEvent(false, msg));
    }
  };

  static class CallbackEvent {
    boolean succeeded=false;
    String msg=null;

    CallbackEvent(boolean succeeded, String msg) {
      this.succeeded=succeeded;
      this.msg=msg;
    }
  }
```

The fragment itself registers for the event bus, and in `onEventMainThread()`, it raises the `Toast`:

```
  public void onEventMainThread(CallbackEvent event) {
    if (getActivity()!=null) {
      if (event.succeeded) {
        Toast.makeText(getActivity(), "Download successful!",
Toast.LENGTH_LONG).show();
      }
      else {
        Toast.makeText(getActivity(), event.msg, Toast.LENGTH_LONG).show();
      }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }
```

And, of course, we need to pass the `IDownloadCallback` object in our `download()` call:

```
@Override
public void onClick(View view) {
  try {
    binding.download(TO_DOWNLOAD, cb);
  }
  catch (RemoteException e) {
    Log.e(getClass().getSimpleName(), "Exception requesting download", e);
    Toast.makeText(getActivity(), e.getMessage(), Toast.LENGTH_LONG).show();
  }
}
```

Revising the Service

The service also needs changing, to use the supplied callback object for the end results of the download.

`DownloadBinder` now receives an `IDownloadCallback` proxy in its `download()` method, which it passes along to the `DownloadThread`:

```
private static class DownloadBinder extends IDownload.Stub {
  @Override
  public void download(String url, IDownloadCallback cb) {
    new DownloadThread(url, cb).start();
  }
}
```

Notice that the service's own API just needs the `IDownloadCallback` parameter, which can be passed around and used like any other Java object. The fact that it happens to cause calls to be made synchronously back to the remote client is invisible to the service.

`DownloadThread`, in turn, invokes `onSuccess()` or `onFailure()` as appropriate:

```
private static class DownloadThread extends Thread {
  String url=null;
  IDownloadCallback cb=null;

  DownloadThread(String url, IDownloadCallback cb) {
    this.url=url;
    this.cb=cb;
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    @Override
    public void run() {
      try {
        File root=

Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);

        root.mkdirs();

        File output=new File(root, Uri.parse(url).getLastPathSegment());

        if (output.exists()) {
          output.delete();
        }

        HttpURLConnection c=(HttpURLConnection)new URL(url).openConnection();

        FileOutputStream fos=new FileOutputStream(output.getPath());
        BufferedOutputStream out=new BufferedOutputStream(fos);

        try {
          InputStream in=c.getInputStream();
          byte[] buffer=new byte[8192];
          int len=0;

          while ((len=in.read(buffer)) >= 0) {
            out.write(buffer, 0, len);
          }

          out.flush();

          try {
            cb.onSuccess();
          }
          catch (RemoteException e) {
            Log.e("DownloadJob", "Exception when calling onSuccess()", e);
          }
        }
        finally {
          fos.getFD().sync();
          out.close();
          c.disconnect();
        }
      }
      catch (IOException e2) {
        Log.e("DownloadJob", "Exception in download", e2);

        try {
          cb.onFailure(e2.getMessage());
        }
        catch (RemoteException e) {
          Log.e("DownloadJob", "Exception when calling onFailure()", e2);
        }
      }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }
```

Thinking About Security

Remote services, by definition, are available for anyone to connect to. This may or may not be a good idea.

If the only client of your remote service is some other app of yours, you could protect the service using a custom signature-level permission.

If you anticipate third-party apps communicating with your service, you should strongly consider protecting the service with an ordinary custom permission, so the *user* can vote on whether the communication is allowed.

For local services, the simplest way to secure the service is to not export it, typically by not having an `<intent-filter>` element for the `<service>` in the manifest. Then, your app is the only app that can work with the service.

The "Everlasting Service" Anti-Pattern

One anti-pattern that is all too prevalent in Android is the "everlasting service". Such a service is started via `startService()` and never stops — the component starting it does not stop it and it does not stop itself via `stopSelf()`.

Why is this an anti-pattern?

1. The service takes up memory all of the time. This is bad in its own right if the service is not continuously delivering sufficient value to be worth the memory.
2. Users, fearing services that sap their device's CPU or RAM, may attack the service with so-called "task killer" applications or may terminate the service via the Settings app, thereby defeating your original goal.
3. Android itself, due to user frustration with sloppy developers, will terminate services it deems ill-used, particularly ones that have run for quite some time.

Occasionally, an everlasting service is the right solution. Take a VOIP client, for example. A VOIP client usually needs to hold an open socket with the VOIP server to know about incoming calls. The only way to continuously watch for incoming

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

calls is to continuously hold open the socket. The only component capable of doing that would be a service, so the service would have to continuously run.

However, in the case of a VOIP client, or a music player, the user is the one specifically requesting the service to run forever. By using `startForeground()`, a service can ensure it will not be stopped due to old age for cases like this.

As a counter-example, imagine an email client. The client wishes to check for new email messages periodically. The right solution for this is the `AlarmManager` pattern described elsewhere in this book. The anti-pattern would have a service running constantly, spending most of its time waiting for the polling period to elapse (e.g., via `Thread.sleep()`). There is no value to the user in taking up RAM to watch the clock tick. Such services should be rewritten to use `AlarmManager`.

Most of the time, though, it appears that services are simply leaked. That is one advantage of using `AlarmManager` and an `IntentService` – it is difficult to leak the service, causing it to run indefinitely. In fact, `IntentService` in general is a great implementation to use whenever you use the command pattern, as it ensures that the service will shut down eventually. If you use a regular service, be sure to shut it down when it is no longer actively delivering value to the user.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Advanced Manifest Tips

If you have been diligent about reading this book (versus having randomly jumped to this chapter), you will already have done a fair number of things with your project's `AndroidManifest.xml` file:

1. Used it to define components, like activities, services, content providers, and manifest-registered broadcast receivers
2. Used it to declare permissions your application requires, or possibly to define permissions that other applications need in order to integrate with your application
3. Used it to define what SDK level, screen sizes, and other device capabilities your application requires

In this chapter, we continue looking at things the manifest offers you, starting with a discussion of controlling where your application gets installed on a device, and wrapping up with a bit of information about activity aliases.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Just Looking For Some Elbow Room

On October 22, 2008, the HTC Dream was released, under the moniker of "T-Mobile G1", as the first production Android device.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Complaints about the lack of available storage space for applications probably started on October 23rd.

The Dream, while a solid first Android device, offered only 70MB of on-board flash for application storage. This storage had to include:

1. The Android application (APK) file
2. Any local files or databases the application created, particularly those deemed unsafe to put on the SD card (e.g., privacy)
3. Extra copies of some portions of the APK file, such as the compiled Dalvik bytecode, which get unpacked on installation for speed of access

It would not take long for a user to fill up 70MB of space, then have to start removing some applications to be able to try others.

Users and developers alike could not quite understand why the Dream had so little space compared to the available iPhone models, and they begged to at least allow applications to install to the SD card, where there would be more room. This, however, was not easy to implement in a secure fashion, and it took until Android 2.2 for the feature to become officially available.

If your app's `android:minSdkVersion` is 11 or higher, you can probably ignore all of this. At that time, what the Android SDK refers to as "internal storage" and "external storage" were moved to be part of one filesystem partition, and so there is no artificial division of space between the two.

But, if you are still supporting Android 2.2 and 2.3, you may wish to consider supporting having your app be installed to, or moved to, external storage.

Configuring Your App to Reside on External Storage

Indicating to Android that your application can reside on the SD card is easy... and necessary, if you want the feature. If you do not tell Android this is allowed, Android will *not* install your application to the SD card, nor allow the user to move the application to the SD card.

All you need to do is add an `android:installLocation` attribute to the root `<manifest>` element of your `AndroidManifest.xml` file. There are three possible values for this attribute:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `internalOnly`, the default, meaning that the application cannot be installed to the SD card
- `preferExternal`, meaning the application would like to be installed on the SD card
- `auto`, meaning the application can be installed in either location

If you use `preferExternal`, then your application will be initially installed on the SD card in most cases. Android reserves the right to still install your application on internal storage in cases where that makes too much sense, such as there not being an SD card installed at the time.

If you use `auto`, then Android will make the decision as to the installation location, based on a variety of factors. In effect, this means that `auto` and `preferExternal` are functionally very similar – all you are doing with `preferExternal` is giving Android a hint as to your desired installation destination.

Because Android decides where your application is initially installed, and because the user has the option to move your application between the SD card and on-board flash, you cannot assume any given installation spot. The exception is if you choose `internalOnly`, in which case Android will honor your request, at the potential cost of not allowing the installation at all if there is no more room in on-board flash.

For example, here is the manifest from the `SMS/Sender` sample project, profiled in another chapter, showing the use of `preferExternal`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.sms.sender"
  android:installLocation="preferExternal"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission android:name="android.permission.READ_CONTACTS"/>
  <uses-permission android:name="android.permission.SEND_SMS"/>

  <uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"/>

  <application
    android:icon="@drawable/ic_launcher"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    android:label="@string/app_name">
    <activity
      android:name="Sender"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Since this feature only became available in Android 2.2, to support older versions of Android, just have your build tools target API level 8 (e.g., `compileSdkVersion` of 8 or higher in `build.gradle` for Android Studio users, or Project > Properties > Android for those of you building with Eclipse) while having your `minSdkVersion` attribute in the manifest state the lowest Android version your application supports overall. Older versions of Android will ignore the `android:installLocation` attribute. So, for example, in the above manifest, the `Sender` application supports API level 4 and above (Android 1.6 and newer), but still can use `android:installLocation="preferExternal"`, because the build tools are targeting API level 8.

What the User Sees

On newer devices, such as those running Android 4.2, the user will see nothing different. That is because internal and external storage share a common pool of space, and therefore there is no advantage in having your application installed to external storage.

However, on, say, Android 2.3, you will see a difference in behavior.

For an application that wound up on external storage, courtesy of your choice of `preferExternal` or `auto`, the user will have an option to move it to the phone's internal storage. This can be done by choosing the application in the Manage Applications list in the Settings application, then clicking the "Move to phone" button.

Conversely, if your application is installed in on-board flash, and it is movable to external storage, they will be given that option with a "Move to SD card" button.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What the Pirate Sees

Ideally, the pirate sees nothing at all.

One of the major concerns with installing applications to the SD card is that the SD card is usually formatted FAT32 (`vfat`), offering no protection from prying eyes. The concern was that pirates could then just pluck the APK file off external storage and distribute it, even for paid apps from the Play Store.

Apparently, they solved this problem.

To quote the Android developer documentation:

> The unique container in which your application is stored is encrypted with a randomly generated key that can be decrypted only by the device that originally installed it. Thus, an application installed on an SD card works for only one device.

Moreover, this "unique container" is not normally mounted when the user mounts external storage on their host machine. The user mounts `/mnt/sdcard`; the "unique container" is `/mnt/asec`.

What Your App Sees… When External Storage is Inaccessible

So far, this has all seemed great for users and developers. Developers need to add a single attribute to the manifest, and Android 2.2+ users gain the flexibility of where the app gets stored.

Alas, there is a problem, and it is a big one: on Android 1.x and 2.x, either the host PC or the device can have access to the SD card, but not both. As a result, if the user makes the SD card available to the host PC, by plugging in the USB cable and mounting the SD card as a drive via a `Notification` or other means, that SD card becomes unavailable for running applications.

So, what happens?

1. First, your application is terminated forcibly, as if your process was being closed due to low memory. Notably, your activities and services will not be called with `onDestroy()`, and instance state saved via `onSaveInstanceState()` is lost.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. Second, your application is unhooked from the system. Users will not see your application in the launcher, your `AlarmManager` alarms will be canceled, and so on.
3. When the user makes external storage available to the phone again, your application will be hooked back into the system and will be once again available to the user (for example, your icon will reappear in the launcher)

The upshot: if your application is simply a collection of activities, otherwise not terribly connected to Android, the impact on your application is no different than if the user reboots the phone, kills your process via a so-called "task killer" application, etc. If, however, you are doing more than that, the impacts may be more dramatic.

Perhaps the most dramatic impact, from a user's standpoint, will be if your application implements app widgets. If the user has your app widget on her home screen, that app widget will be removed when the SD card becomes unavailable to the phone. Worse, your app widget cannot be re-added to the home screen until the phone is rebooted (a limitation that hopefully will be lifted sometime after Android 2.2).

The user is warned about this happening, at least in general:

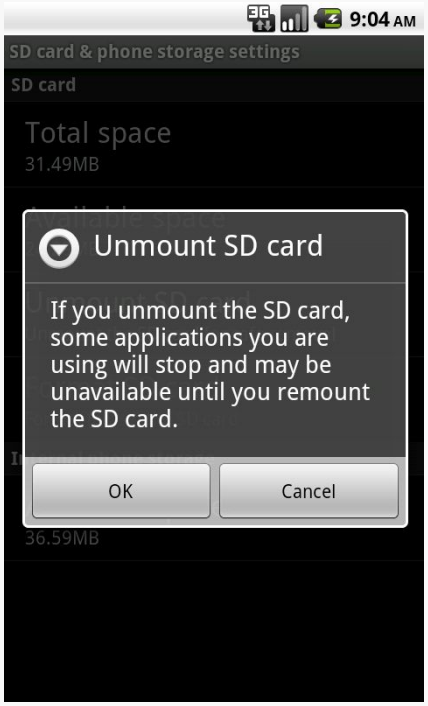


Figure 701: Warning when unmounting the SD card

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Two broadcast Intents are sent out related to this:

- `ACTION_EXTERNAL_APPLICATIONS_UNAVAILABLE`, when the SD card (and applications installed upon it) become unavailable
- `ACTION_EXTERNAL_APPLICATIONS_AVAILABLE`, when the SD card and its applications return to normal

Note that the documentation is unclear as to whether your own application, that had been on the SD card, can receive `ACTION_EXTERNAL_APPLICATIONS_AVAILABLE` once the SD card is back in action. There is an outstanding issue on this topic in the Android issue tracker.

Also note that all of these problems hold true for longer if the user physically removes the SD card from the device. If, for example, they replace the card with a different one — such as one with more space — your application will be largely lost. They will see a note in their applications list for your application, but the icon will indicate it is on external storage, and the only thing they can do is uninstall it:

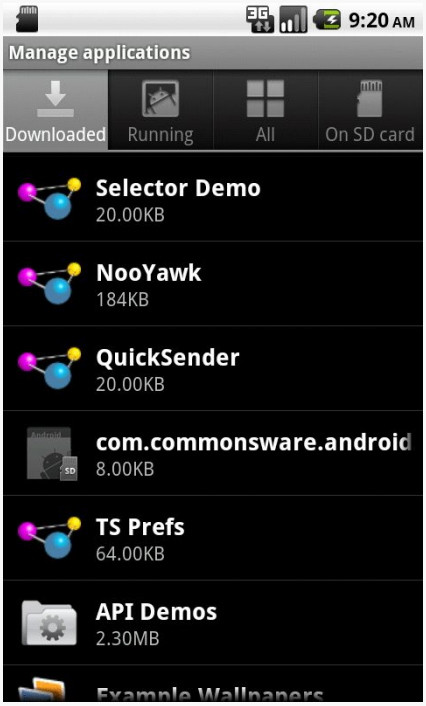


Figure 702: The Manage Applications list, with an application shown from a removed SD card

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Choosing Whether to Support External Storage

Given the huge problem from the previous section, the question of whether or not your application should support external storage is far from clear.

As the Android developer documentation states:

> Large games are more commonly the types of applications that should allow installation on external storage, because games don't typically provide additional services when inactive. When external storage becomes unavailable and a game process is killed, there should be no visible effect when the storage becomes available again and the user restarts the game (assuming that the game properly saved its state during the normal Activity lifecycle).

Conversely, if your application implements any of the following features, it may be best to not support external storage:

1. Polling of Web services or other Internet resources via a scheduled alarm
2. Account managers and their corresponding sync adapters, for custom sources of contact data
3. App widgets, as noted in the previous section
4. Device administration extensions
5. Live folders
6. Custom soft keyboards ("input method engines")
7. Live wallpapers
8. Custom search providers

But, as noted earlier, this is not even usually necessary on API Level 11+ devices. Hence, even if your app would otherwise qualify for being installed to external storage, you may not wish to bother. If few devices (Android 2.2 and Android 2.3) might need the capability, it may not be worth the extra testing burden.

Using an Alias

As was mentioned in the chapter on integration, you can use the `PackageManager` class to enable and disable components in your application. This works at the component level, meaning you can enable and disable activities, services, content providers, and broadcast receivers. It does not support enabling or disabling individual `<intent-filter>` stanzas from a given component, though.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Why might you want to do this?

1. Perhaps you have an activity you want to be available for use, but not necessarily available in the launcher, depending on user configuration or unlocking "pro" features or something
2. Perhaps you want to add browser support for certain MIME types, but only if other third-party applications are not already installed on the device

While you cannot control individual `<intent-filter>` stanzas directly, you can have a similar effect via an activity alias.

An activity alias is another manifest element — `<activity-alias>` – that provides an alternative set of filters or other component settings for an already-defined activity. For example, here is the `AndroidManifest.xml` file from the Manifest/Alias sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.alias"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <supports-screens android:largeScreens="true"
                    android:normalScreens="true"
                    android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name="AliasActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <activity-alias android:label="@string/app_name2"
                    android:name="ThisIsTheAlias"
                    android:targetActivity="AliasActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity-alias>
  </application>
</manifest>
```

Here, we have one `<activity>` element, with an `<intent-filter>` to put the activity in the launcher. We also have an `<activity-alias>` element... which puts a second icon in the launcher for the same activity implementation.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

An activity alias can be enabled and disabled independently of its underlying activity. Hence, you can have one activity class have several independent sets of intent filters and can choose which of those sets are enabled at any point in time.

For testing purposes, you can also enable and disable these from the command line. Use the `adb shell pm disable` command to disable a component:

```
adb shell pm disable
com.commonsware.android.alias/com.commonsware.android.alias.ThisIsTheAlias
```

... and the corresponding `adb shell pm enable` command to enable a component:

```
adb shell pm enable
com.commonsware.android.alias/com.commonsware.android.alias.ThisIsTheAlias
```

In each case, you supply the package of the application (`com.commonsware.android.alias`) and the class of the component to enable or disable (`com.commonsware.android.alias.ThisIsTheAlias`), separated by a slash.

Getting Meta (Data)

Sometimes, you may want to put more data in the manifest, associated with your components. You will frequently see this for use with libraries or plugin distribution models, where sharing some configuration data between parties could eliminate a bunch of API code that a reuser might need to implement.

To support this, Android offers a `<meta-data>` element as a child of `<activity>`, `<activity-alias>`, `<receiver>`, or `<service>`. Each `<meta-data>` element has an `android:name` attribute plus an associated value, supplied by either an `android:value` attribute (typically for literals) or an `android:resource` attribute (for references to resources).

Other parties can then get at this information via `PackageManager`. So, for example, the implementer of a plugin could have `<meta-data>` elements indicating details of how the plugin should be used (e.g., desired polling frequency), and the host of the plugin could then get that configuration data without the plugin author having to mess around with implementing some Java API for it.

For example, Roman Nurik's DashClock is a lockscreen app widget designed to serve as a replacement for the clock app widget that ships with many Android 4.2+ devices. Not only does it display the time, but it is a plugin host, allowing third party

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

developers to supply "extensions" that can also display data in the app widget. This way, users can set up a single lockscreen app widget and get at a bunch of useful information.

DashClock's extension API makes use of `<meta-data>` to pass configuration data from the extension to DashClock itself. The implementation of a DashClock extension is a service, and so the extension's `<service>` element will have a batch of `<meta-data>` elements with this configuration data:

```
<service android:name=".ExampleExtension"
     android:icon="@drawable/ic_extension_example"
     android:label="@string/extension_title"

android:permission="com.google.android.apps.dashclock.permission.READ_EXTENSION_DATA">
     <intent-filter>
         <action android:name="com.google.android.apps.dashclock.Extension" />
     </intent-filter>
     <meta-data android:name="protocolVersion" android:value="1" />
     <meta-data android:name="description"
         android:value="@string/extension_description" />
     <!-- A settings activity is optional -->
     <meta-data android:name="settingsActivity"
         android:value=".ExampleSettingsActivity" />
 </service>
```

(sample from the DashClock documentation)

Here, the developer can specify:

- What version of the communications protocol is supported, so DashClock can update its protocol over time yet remain backwards-compatible with older extensions, via the `protocolVersion` entry
- What the description is for the extension, used in DashClock's configuration screens to let the user know what available extensions there are, via the `description` entry
- What activity, if any, does the extension supply that allows the user to configure that extension, that DashClock should provide access to from its own settings activity, via the `settingsActivity` entry

In all three cases, DashClock uses `android:value`. Note that `android:value` *does* support the use of resources — the value of `description` is a reference to the `extension_description` string resource, for example.

To retrieve that metdata, an app can ask for `PackageManager.GET_META_DATA` as a flag on `PackageManager` methods for introspection, like `queryIntentActivities()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In the case of DashClock, it retrieves all implementations of its plugin by asking Android what *services* have an `<intent-filter>` with an `<action>` of `com.google.android.apps.dashclock.Extension`, via `queryIntentServices()`, asking for `PackageManager` to also supply each service's metadata:

```
List<ResolveInfo> resolveInfos = pm.queryIntentServices(
                new Intent(DashClockExtension.ACTION_EXTENSION),
PackageManager.GET_META_DATA);
```

(from the `ExtensionManager.java` file in the DashClock source code)

Each `ResolveInfo` object that comes back in the list will have a `serviceInfo` field containing details of the service. Because `GET_META_DATA` was passed in as a flag, the `serviceInfo` will have a `Bundle` named `metaData` which will contain the key/value pairs specified by the `<meta-data>` elements. DashClock can then grab that data and use it to populate its own object model:

```
for (ResolveInfo resolveInfo : resolveInfos) {
    ExtensionListing listing = new ExtensionListing();
    listing.componentName = new
ComponentName(resolveInfo.serviceInfo.packageName,
            resolveInfo.serviceInfo.name);
    listing.title = resolveInfo.loadLabel(pm).toString();
    Bundle metaData = resolveInfo.serviceInfo.metaData;
    if (metaData != null) {
        listing.protocolVersion = metaData.getInt("protocolVersion");
        listing.description = metaData.getString("description");
        String settingsActivity = metaData.getString("settingsActivity");
        if (!TextUtils.isEmpty(settingsActivity)) {
            listing.settingsActivity = ComponentName.unflattenFromString(
                    resolveInfo.serviceInfo.packageName + "/" +
settingsActivity);
        }
    }
```

(from the `ExtensionManager.java` file in the DashClock source code)

The `<meta-data>` element supports five data types for `android:value`:

- String
- Integer
- Boolean (specified as `true` or `false` in the `android:value` attribute)
- Float

It also supports colors, specified in #AARRGGBB and similar formats, which, according to the documentation, is returned as a string.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In this fashion, extension developers can supply enough information for DashClock to allow the user to see the list of installed extensions, choose which one(s) they want, and configure those (where applicable). Actually getting the *content* to display will need to be done at runtime, in this case via making requests of the service to supply a `ExtensionData` structure with the messages, icon, and so forth to be displayed.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Miscellaneous Integration Tips

This chapter is a collection of other miscellaneous integration and introspection tips and techniques that you might find useful in your Android apps.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Take the Shortcut

Another way to integrate with Android is to offer custom shortcuts. Shortcuts are available from the home screen. Whereas app widgets allow you to draw on the home screen, shortcuts allow you to wrap a custom Intent with an icon and caption and put that on the home screen. You can use this to drive users not just to your application's "front door", like the launcher icon, but to some specific capability within your application, like a bookmark.

In our case, in the `Introspection/QuickSender` sample project, we will allow users to create shortcuts that use `ACTION_SEND` to send a pre-defined message, either to a specific address or anywhere, as we have seen before in this chapter.

Once again, the key is in the intent filter.

Registering a Shortcut Provider

Here is the manifest for `QuickSender`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.qsender"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <supports-screens android:largeScreens="true"
                    android:normalScreens="true"
                    android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name="QuickSender">
      <intent-filter>
        <action android:name="android.intent.action.CREATE_SHORTCUT" />
        <category android:name="android.intent.category.DEFAULT" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Our single activity does not implement a traditional launcher `<intent-filter>`. Rather, it has one that watches for a `CREATE_SHORTCUT` action. This does two things:

- It means that our activity will show up in the list of possible shortcuts a user can configure
- It means this activity will be the recipient of a `CREATE_SHORTCUT` Intent if the user chooses this application from the shortcuts list

Implementing a Shortcut Provider

The job of a shortcut-providing activity is to:

1. Create an `Intent` that will be what the shortcut launches
2. Return that `Intent` and other data to the activity that started the shortcut provider
3. Finally, `finish()`, so the caller gets control

You can see all of that in the `QuickSender` implementation:

```
package com.commonsware.android.qsender;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.text.TextUtils;
import android.view.View;
import android.widget.TextView;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public class QuickSender extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void save(View v) {
    Intent shortcut=new Intent(Intent.ACTION_SEND);
    TextView addr=(TextView)findViewById(R.id.addr);
    TextView subject=(TextView)findViewById(R.id.subject);
    TextView body=(TextView)findViewById(R.id.body);
    TextView name=(TextView)findViewById(R.id.name);

    if (!TextUtils.isEmpty(addr.getText())) {
      shortcut.putExtra(Intent.EXTRA_EMAIL,
                        new String[] { addr.getText().toString() });
    }

    if (!TextUtils.isEmpty(subject.getText())) {
      shortcut.putExtra(Intent.EXTRA_SUBJECT, subject.getText()
                                                     .toString());
    }

    if (!TextUtils.isEmpty(body.getText())) {
      shortcut.putExtra(Intent.EXTRA_TEXT, body.getText().toString());
    }

    shortcut.setType("text/plain");

    Intent result=new Intent();

    result.putExtra(Intent.EXTRA_SHORTCUT_INTENT, shortcut);
    result.putExtra(Intent.EXTRA_SHORTCUT_NAME, name.getText()
                                                    .toString());
    result.putExtra(Intent.EXTRA_SHORTCUT_ICON_RESOURCE,
                    Intent.ShortcutIconResource.fromContext(this,
                                                            R.drawable.icon));

    setResult(RESULT_OK, result);
    finish();
  }
}
```

The `shortcut` `Intent` is the one that will be launched when the user taps the shortcut icon on the home screen. The `result` `Intent` packages up shortcut plus the icon and caption, where the icon is converted into an `Intent.ShortcutIconResource` object. That `result` `Intent` is then used with the `setResult()` call, to pass that back to whatever called `startActivityForResult()` to open up `QuickSender`. Then, we `finish()`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

At this point, all the information about the shortcut is in the hands of Android (or, more accurately, the home screen application), which can add the icon to the home screen.

Using the Shortcuts

To create a custom shortcut using `QuickSender`, long-tap on the background of the home screen to bring up the customization options:

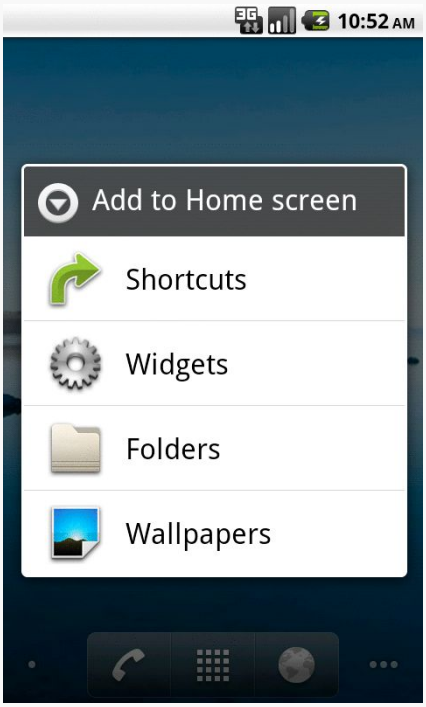


Figure 703: The home screen customization options list

Choose Shortcuts, and scroll down to find `QuickSender` in the list:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

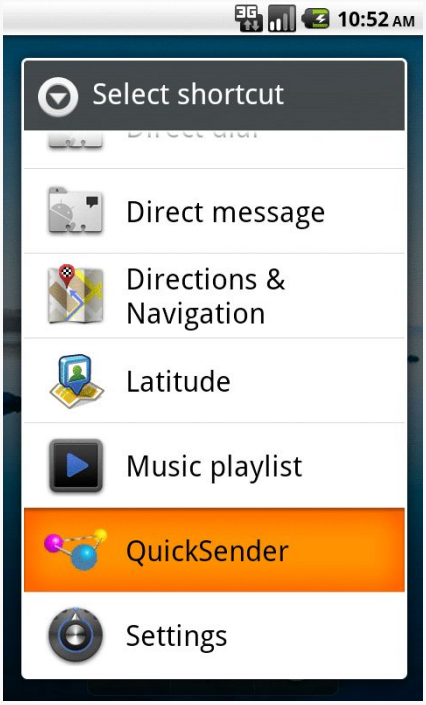


Figure 704: The available types of shortcuts

Click the `QuickSender` entry, which will bring up our activity with the form to define what to send:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

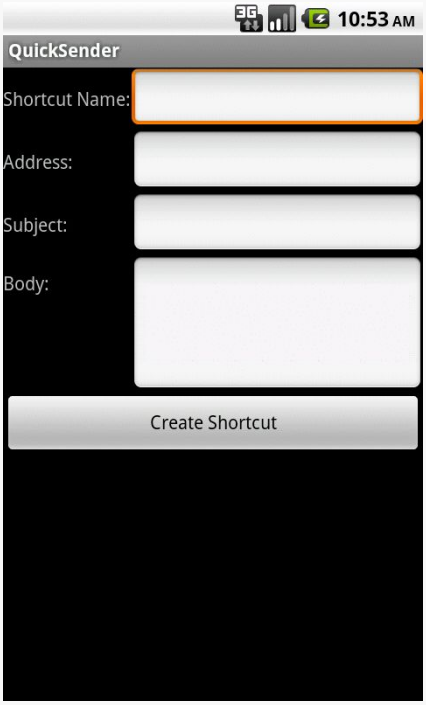


Figure 705: The QuickSender configuration activity

Fill in the name, either the subject or body, and optionally the address. Then, click the Create Shortcut button, and you will find your shortcut sitting on your home screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

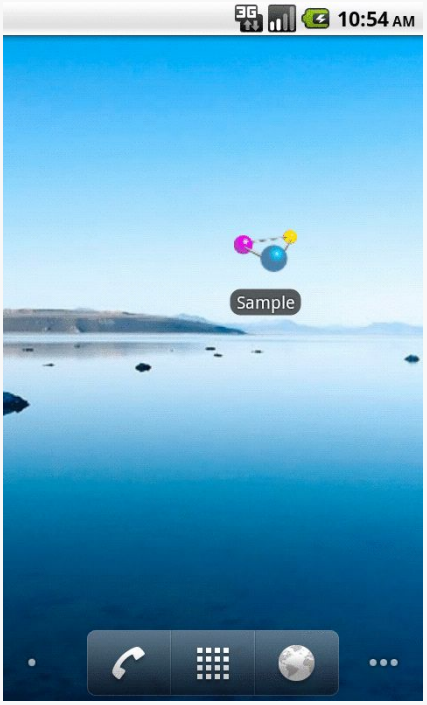


Figure 706: The QuickSender-defined shortcut, labeled Shortcut

If you launch that shortcut, and if there is more than one application on the device set up to handle `ACTION_SEND`, Android will bring up a special chooser, to allow you to not only pick how to send the message, but optionally make that method the default for all future requests:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 707: The ACTION_SEND request, as triggered by the shortcut

Depending on what you choose, of course, will dictate how the message actually gets sent.

Homing Beacons for Intents

If you are encountering problems with `Intent` resolution — you create an `Intent` for something and try starting an `Activity` or `Service` with it, and it does not work — you can add the `FLAG_DEBUG_LOG_RESOLUTION` flag to the `Intent`. This will dump information to LogCat about how the `Intent` resolution occurred, so you can better diagnose what might be going wrong.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Reusable Components

In the world of Java outside of Android, reusable components rule the roost. Whether they are simple JARs, are tied in via inversion-of-control (IoC) containers like Spring, or rely on enterprise service buses like Mule, reusable Java components are a huge portion of the overall Java ecosystem. Even full-fledged applications, like Eclipse or NetBeans, are frequently made up of a number of inter-locking components, many of which are available for others to use in their own applications.

Android, too, supports this sort of reuse. In some cases, it follows standard Java approaches. However, in other cases, unique Android aspects, such as resources, steer developers in different directions for reuse.

This chapter will outline what reuse models are in use today and how you can package your own components for reuse.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Where Do I Find Them?

Android historically has not had a "go-to" place to find reusable components. The Android Arsenal is probably the largest collection at present. Beyond that, look for recommendations in Stack Overflow answers, blog posts, and the like.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

How Are They Packaged?

There are three main ways that reusable code gets packaged on Android: as a traditional Java JAR, as an Android library project, or (technically) as an APK. The last approach is usually used by apps that have user value in their own right, but also expose some sort of integration API for use by other apps, that you can take advantage of.

JARs

Android code that is pure *code*, without requiring its own resources, can be packaged into a JAR, no differently than can regular Java code outside of Android.

As was covered earlier in the book, to use such a JAR, just drop it into `libs/`. Its contents will be added to your compile path (so you can reference classes from the library) and its contents will be packaged in your APK (so those references will work at runtime).

Library Projects

Android code that relies upon resources — such as many reusable UI components, such as custom widgets — cannot be packaged as a simple JAR, as there is no way of packaging the Android resources in that JAR. Instead, Google created the Android library project as the "unit of reuse" for such cases.

Android library projects are sometimes published in full source form (usually open source projects), and sometimes are published as AARs in an artifact repository. Eclipse users can readily use the full-source library projects, but have limited ability to use AARs. Android Studio users can use either, and AARs may be as simple as adding a single line to `build.gradle`.

APKs

Using JARs or library projects fits in the "traditional" model of compile-time reuse. Android's many IPC mechanisms offer plenty of options for run-time reuse, where your app communicates with another app, having that app do things on your behalf. In this case, the primary unit of reuse is not the JAR, or the library project, but the APK.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For example, the ZXing project publishes the Barcode Scanner app. This app not only allows *users* to scan barcodes, but allows *other apps* to scan barcodes, by asking Barcode Scanner to scan the barcodes and return results.

To integrate with such an app, you will need to find the instructions from the app's developers on how to do that. Sometimes, they will tell you things that you would use directly (e.g., "call `startActivityForResult()` with an `Intent` that contains..."). Sometimes, they will distribute a client-side JAR that you can use that wraps up the low-level IPC details into something a bit easier to consume. For example, ZXing distributes an `IntentIntegrator.java` class file that you can use that not only handles requesting the scans, but also helping the user install Barcode Scanner if it is not already installed.

How Do I Create Them?

To create a reusable component, you start by getting a working code base, one that implements whatever it is that you desire. From there, you need to choose which of those aforementioned distribution patterns you believe is appropriate:

- JAR
- Standard library project
- Eclipse-compatible binary-only library project
- APK (with optional client-side JAR)

That, in turn, will drive how you take your code and create such a package. The basics of how to do that for the different alternatives is described in the following sections.

JARs

Creating a JAR for a reusable chunk of Android-related code is not significantly different than is creating a JAR for a reusable chunk of "ordinary" Java code.

First, you need a project that represents the "resuable chunk of Android-related code". An easy way to do this is to just create a standard Android library project, but one where you do not bother creating any resources.

Once the code is ready for distribution, you can create a JAR from the compiled Java classes by your favorite traditional means. The author of this book, for example, adds custom Gradle tasks for this:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
// from http://stackoverflow.com/a/19484146/115145

android.libraryVariants.all { variant -> def name = variant.buildType.name if
(name.equals(com.android.builder.core.BuilderConstants.DEBUG)) { return; // Skip
debug builds. } def task = project.tasks.create "jar${name.capitalize()}", Jar
task.dependsOn variant.javaCompile task.from variant.javaCompile.destinationDir
task.archiveName = "cwac-${task.archiveName}" task.exclude('com/commonsware/
cwac//BuildConfig.') }
```

This will create JAR-building Gradle tasks for all non-debug build types, so you get Gradle tasks like `jarRelease`. It specifically excludes `BuildConfig`, which the CWAC libraries never use, but otherwise takes all of the library classes and packages them in a JAR, named after the library and version, with a `cwac-` prefix.

If your reusable code is pure Java, not involving Android at all, you are welcome to create a plain Java project and create your JAR from that. The only major recommendation would be to ensure that you are using some `android.jar` from the SDK, rather than a JDK `rt.jar`, to ensure that you are sticking with classes and methods that are in Android's subset of the Java SE class library.

Standard Library Projects

In many respects, distributing a standard Android library project is even easier: just ZIP it up. Or, if it is in a public source control repository (e.g., GitHub), reusers can obtain it from that repository.

Of course, this will distribute the source code along with the resources and everything else. This is typical for an open source library project.

Android Studio and Gradle users can create AARs from their library projects. The `assembleRelease` task will create an AAR for the library in `build/outputs/aar`, named after the library and version (e.g., `pager-0.2.3.jar`).

Eclipse-Compatible Binary-Only Library Projects

AARs do not ship Java source code, but rather only binaries. However, AARs are not readily consumable from Eclipse.

It is possible to create an Eclipse-Compatible binary-only library project, one where your source code is replaced by a JAR. This can be useful for proprietary library

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

projects, for example. However, there is one noteworthy limitation with today's tools: the library project cannot itself depend upon a JAR or another library project.

For simpler library projects, the recipe is straightforward, given an already-existing Android library project:

1. Compile the Java source (e.g., via Ant) and turn it into a JAR file.
2. Create a copy of your original Android library project to serve as a distribution Android library project.
3. Place the compiled JAR from step #1 and put it in `libs/` of the distribution library project from step #2.
4. Delete everything in src/ of the distribution library project (but leave the now-empty `src/` directory there).
5. Distribute the distribution library project (e.g., ZIP it up)

For example, an Ant target to create a distribution ZIP might be:

```
<target name="jar" depends="release">
    <delete file="bin/WhateverYouWantToCallYourLibrary.jar" />
    <jar destfile="bin/WhateverYouWantToCallYourLibrary.jar">
      <fileset dir="bin/classes">
        <exclude name="**/BuildConfig.class" />
        <exclude name="**/R.class" />
        <exclude name="**/R$*.class" />
      </fileset>
    </jar>
</target>
<target name="dist" depends="jar">
  <copy todir="/tmp/WhateverYouWantToCallYourLibrary/libs">
    <fileset dir="libs/" />
  </copy>
  <copy todir="/tmp/WhateverYouWantToCallYourLibrary/res">
    <fileset dir="res/" />
  </copy>
  <copy
    file="bin/WhateverYouWantToCallYourLibrary.jar"
    todir="/tmp/WhateverYouWantToCallYourLibrary/libs" />
  <copy
    file="AndroidManifest.xml"
    todir="/tmp/WhateverYouWantToCallYourLibrary" />
  <copy file="build.xml" todir="/tmp/WhateverYouWantToCallYourLibrary" />
  <copy
    file="project.properties"
    todir="/tmp/WhateverYouWantToCallYourLibrary" />
  <copy file="LICENSE" todir="/tmp/WhateverYouWantToCallYourLibrary" />
  <mkdir dir="/tmp/WhateverYouWantToCallYourLibrary/src" />
  <zip
    destfile="/tmp/WhateverYouWantToCallYourLibrary.zip"
    basedir="/tmp/"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    includes="WhateverYouWantToCallYourLibrary/**"
    whenempty="create" />
  <delete dir="/tmp/WhateverYouWantToCallYourLibrary" />
</target>
```

Assuming the existence of a `/tmp/` directory (e.g., OS X or Linux), this will result in a `WhateverYouWantToCallYourLibrary.zip` file in `/tmp/`. Along the way, we:

- Copy the `libs/` and `res/` trees from your source library project to a temporary distribution directory
- Copy your compiled JAR into the `libs/` subdirectory of the temporary distribution directory
- Copy other miscellaneous files, like your `LICENSE` file for your software license terms, into the root of the temporary distribution directory
- Create an empty `src/` subdirectory in the temporary distribution directory
- ZIP up the temporary distribution directory to a ZIP file
- Delete the temporary distribution directory

APK

Most of your work for this distribution model is in writing and distributing the app to your end users, through the Play Store or your other chosen distribution channels.

In addition to that, you need to either document to reusers what sorts of IPC your app supports, or create a JAR or library project that reusers can use to perform that sort of integration. In the latter case, you would have a separate project representing that JAR or library project that you would distribute using any of the aforementioned approaches.

Other Considerations for Publishing Reusable Code

Of course, there is more to publishing a resuable component than code and perhaps Android resources. The following sections outline some other things to consider as you contemplate offering some code base up for reuse by third parties.

Licensing

Your reusable code should be accompanied by adequate licensing information.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Your License

The first license you should worry about is your own. Is your component open source? If so, you will want to ship a license file containing those terms. If your component is not open source, make sure there is a license agreement shipped with the component that lets the reuser know the terms of use.

Bear in mind that not all of your code necessarily has to have the same license. For example, you might have a proprietary license for the component itself, but have sample code be licensed under Apache License 2.0 for easy copy-and-paste.

Third-Party License Impacts

You may need to include licenses for third party libraries that you have to ship along with your own JAR. Obviously, those licenses would need to give you redistribution rights — otherwise, you cannot ship those libraries in the first place.

Sometimes, the third party licenses will impact your project more directly, such as:

1. Incorporating a GPL library may require your project to be licensed under the same license
2. Adding support for Facebook data may require you to limit your API or require reusers to supply API access keys, since you probably do not have rights to redistribute Facebook data

Documenting the Usage

If you are expecting people to reuse your code, you are going to have to tell them how to do that. Usually, these sorts of packages ship documentation with them, sometimes a clone of what is available online. That way, developers can choose the local or hosted edition of the documentation as they wish.

Note that generated documentation (e.g., Javadocs) may still need to be shipped or otherwise supplied to reusers, if you are not providing the source code in the package. Without the source code, reusers cannot regenerate the Javadocs.

Many open source projects eschew formal documentation in favor of simple JavaDocs, plus “documentation in the form of a test suite” or “documentation in the form of sample apps”. While test suites and sample apps are useful supplements, they are not always an effective replacement for written documentation. And, while

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

JavaDocs are useful for reference material, they are often difficult to comprehend for those trying to get started with the code and not knowing where to begin.

Naming Conventions

Make sure that your Java code is in a package that is likely to be distinct from any others that reusers might already have. Typically, this means that the package name is based on a domain name that you control, much like the package name for Android apps themselves. Whatever you do, please do not publish your own code as `android.*`, unless you are contributing this code to the Android open source project, as `android.*` is reserved for use by Android itself.

(The author of this book would also appreciate it if you would not use `com.commonsware.*`)

Also, be careful about the names of your resources. While your Java code resides in its own namespace, your resources are pooled with all other resources in use by the app. As a result, if you decide to reference `R.layout.main` thinking that it will be *your* `main.xml` layout resource, it might actually be replaced by a `main.xml` resource written by the app developer. You may wish to use some sort of a prefix convention on your resource names to reduce the odds of accidental collision:

- ActionBarSherlock uses `abs__`
- ViewPagerIndicator uses `vpi__`
- And so on

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Scripting Languages

Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Role of Scripting Languages

A scripting language, for the purpose of this book, has two characteristics:

1. It is interpreted from source and so does not require any sort of compilation step
2. It cannot (presently) be used to create a full-fledged Android application without at least some form of custom Java-based stub, and probably much more than that

In this part of the book, we will look at scripting languages on Android and what you can accomplish with them, despite any limitations inherent in their collective definition.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

All Grown Up

Interpreted languages have been a part of the programming landscape for decades. The language most associated with the desktop computer revolution — BASIC — was originally an interpreted language. However, the advent of MS-DOS and the IBM PC (and clones) led developers in the direction of C for "serious programming", for reasons of speed. While interpreted languages continued to evolve, they tended to be described as "scripting" languages, used to glue other applications together. Perl, Python, and the like were not considered "serious" contenders for application development.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The follow-on revolution, for the Internet, changed all of that. Most interactive Web sites were written as CGI scripts using these "toy" languages, Perl first and foremost. Even in environments where Perl was unpopular, such as Windows, Web applications were still written using scripting languages, such as VBScript in Active Server Pages (ASP). While some firms developed Web applications using C/C++, scripting languages ruled the roost. That remains to this day, where you are far more likely to find people writing Web applications in PHP or Ruby than you will find them writing in C or C++. The most likely compiled language for Web development — Java — is still technically an interpreted language, albeit not usually considered a scripting language.

Nowadays, writing major components of an application using a scripting language is not terribly surprising. While this is still most common with Web applications, you can find scripting languages used in the browser (JavaScript), games (Lua), virtual worlds (LSL), and so on. Even though these languages execute more slowly than their C/C++ counterparts, they offer much greater flexibility, and faster CPUs make the performance of scripts less critical.

Following the Script

Scripting languages are not built into Android, beyond the JavaScript interpreter in the WebKit Web browser. Despite this, there is quite a bit of interest in scripting on Android, and the biggest reasons for this come down to experience and comfort level.

Your Expertise

Perhaps you have spent your entire career writing Python scripts, or you cut your teeth on Perl CGI programs, or you have gotten seriously into Ruby development.

Maybe you used Java in previous jobs and hate it with the fiery passion of a thousand suns.

Regardless of the cause, your expertise may lie outside the traditional Android realm of Java-based development. Perhaps you would never touch Android if you had to write in Java, or maybe you feel you would just be significantly more productive in some other language. How much that productivity gain is real versus "in your head" is immaterial — if you want to develop in some other language, you owe it to yourself to try.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Your Users' Expertise

Maybe you are looking to create a program where not only you can write scripts, but so can your users. This might be a utility, or a game, or rulesets for email management, or whatever.

In that case, you need:

1. Something interpreted, so you can execute what the user types in
2. Something embeddable, so your larger application (typically written in Java, of course) is capable of executing those scripts
3. Something your users will be comfortable using for scripting

The last criterion is perhaps the toughest, as non-developers typically have limited experience in writing scripts in any language, let alone one that runs on Android. Perhaps the most popular such language is Basic, in the form of VBA and VBScript on Windows... but there are no interpreters for those languages for Android at this time.

Crowd-Developing

Perhaps your users will not only be entering scripts for their own benefit, but for others' benefit as well.

Many platforms have been improved by power users and amateur developers alike. Browser users gain from those writing GreaseMonkey scripts. Bloggers benefit from those writing WordPress themes. And so on.

To facilitate this sort of work, not only do you need an interpreted, embeddable, user-familiar scripting environment, but you need some means for users to publish their scripts and download the scripts of others. Fortunately, with Android having near-continuous connectivity, your challenge will lie more on organizing and hosting the scripts, more so than getting them on and off of devices.

Going Off-Script

Scripting languages on Android have their fair share of issues. It is safe to say that while Android does not prohibit the use of scripting languages, its architecture does not exactly go out of its way to make them easy to use, either.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Security

For a scripting language to do much that is interesting, it is going to need some amount of privileges. A script cannot access the Internet unless its process has that right. A script cannot modify the user's contacts unless its process has that right. And so on.

For scripts you write, so long as those scripts cannot be modified readily by malware authors, security is whatever you define it to be. If your script-based application needs Internet access, so be it.

For scripts your users write, things get a bit more challenging, since permissions cannot be modified on the fly by applications. Many interpreters will tend to request (or otherwise have access to) permissions that are broader than any individual user might need, because those permissions are needed by somebody. However, the risk is still minimal to the user, so long as they are careful with the scripts they write.

For scripts your users might download, written by others, security becomes a big problem. If the interpreter has a wide range of permissions, downloaded scripts can easily host malware that exploits those permissions for nefarious ends. An interpreter with both Internet access and the right to read the user's contacts means that any script the user might download and run could copy the user's contact data and send it to spammers or identity thieves.

Performance

Java, as interpreted by the Dalvik virtual machine, is reasonably fast, particularly on Android 2.2 and newer versions. C/C++, through the NDK, is far faster.

Scripting languages are a mixed bag.

Some scripting languages for Android have interpreters that are implemented in C code. Those interpreters' performance is partly a function of how well they were written and ported over to the chipsets Android runs on. However, if those interpreters expose Android APIs to the language, that can add considerable overhead. For example, the Scripting Layer for Android (SL4A) makes Android APIs available to scripting languages via a tiny built-in Java Web server and a Web service API. While convenient for language integration, converting simple Java calls into Web service calls slows things down quite a bit.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Some scripting languages have interpreters that themselves are written in Java and run on the virtual machine. Those are likely to perform worse on an Android device than when they are run on a desktop or server, simply because of the performance differences between the standard Java VMs and the Dalvik VM. However, they will have quicker access to the Java class libraries that make up much of Android than will C-based interpreters.

Cross-Platform Compatibility

Most of the scripting languages for Android are ports from versions that run across multiple platforms. This is one of their big benefits – that is where you and your users may have gained experience with those languages. However, just as, say, Perl and Python run a bit differently on Windows than on Linux or OS X, there will be some differences in how those languages run on Android. The Android operating system is not a traditional Linux environment, and so file paths, environment variables, available pre-installed programs, and the like will not be the same. Some of those may, in turn, impact how the scripting languages operate. You may need to make some modification to any existing scripts for those languages that you attempt to run on Android.

Maturity… On Android

Some scripting languages that have been ported to Android are rather old, like Perl and Python. Others are old and somewhat abandoned for traditional development, like BeanShell. Yet others are fairly new to the programming scene altogether, like JRuby.

However, none of them have a long track record on Android, simply because Android itself has not been around very long. This has several implications:

1. There is more likely to be bugs in newer ports of a language than older ports
2. Fewer people will have experience in supporting these languages on Android (compared to supporting them on Linux, for example)
3. The number of production applications built using these languages on Android is minuscule compared to their use on more traditional environments

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Scripting Layer for Android

When it comes to scripting languages on Android, the first stop should always be the Scripting Layer for Android (SL4A). Led by Damon Kohler, this project is rather popular, both among hardcore Android developers and those people looking to automate a bit more of their Android experience.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

The Role of SL4A

What started as an experiment to get Python and Lua going on Android, back in late 2008, turned into a more serious endeavor in June 2009, when the Android Scripting Environment (now called the Scripting Layer for Android, or SL4A) was announced on the Google Open Source blog and the Google Code site for it was established. Since then, SL4A has been a magnet for people interested in getting their favorite language working on Android or advancing its support.

On-Device Development

Historically, the primary role of SL4A was as a tool to allow people to put together scripts, often written on the device itself, to take care of various chores. This appealed to developers who were looking for something lightweight compared to the Android SDK and Java. For those used to tinkering with scripts on other mobile Linux platforms (e.g., the Nokia N800 running Maemo), SL4A promised a similar sort of capability.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Over time, SL4A's scope in this area has grown, including preliminary support for SL4A scripts packaged as APK files, much like an Android application written in Java or any of the alternative frameworks described in this book.

Getting Started with SL4A

SL4A is a bit more difficult to install than is the average Android application, due to the various interpreters it uses and their respective sizes. That being said, none of the steps involved with getting SL4A set up are terribly difficult, and most are just part of the application itself.

Installing SL4A

At the time of this writing, SL4A is not distributed via the Android Market. Instead, you can download it to your device off of the SL4A Web site. Perhaps the easiest way to do that is to scan the QR code on the SL4A home page using Barcode Scanner or a similar utility.

Installing Interpreters

When you first install SL4A, the only available scripting language is for shell scripts, as that is built into Android itself. If you want to work with other interpreters, you will need to download those. That is why the base SL4A download is so small (~200KB) — most of the smarts are separate downloads, largely due to size.

To add interpreters, launch SL4A from the launcher, then choose View > Interpreters from the option menu. You will be presented with the (presently short) list of installed interpreters:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

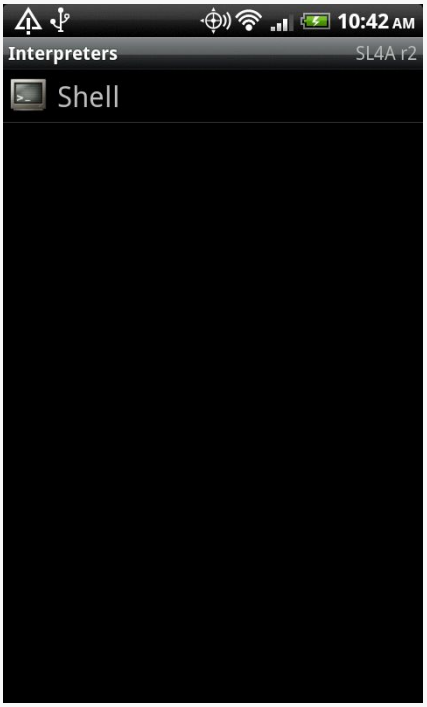


Figure 708: The initial list of installed SL4A interpreters

Then, to install additional interpreters, choose Add from the option menu. You will be given a roster of SL4A-compatible interpreters to choose from:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

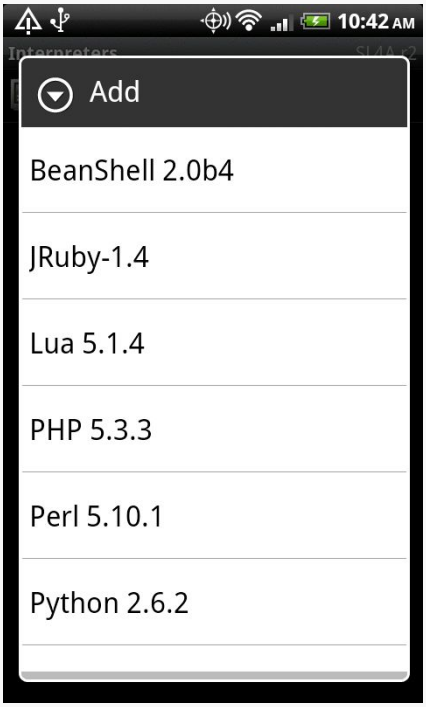


Figure 709: The list of available SL4A interpreters

Click on one of the interpreters, and this will trigger the download of an APK file for that specific interpreter. Slide down the notification drawer and click on that APK file to continue the installation process. When the APK itself is installed, open up that interpreter (e.g., click the "Open" button when the install is done). That will bring up an activity to let you download the rest of the interpreter binaries:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

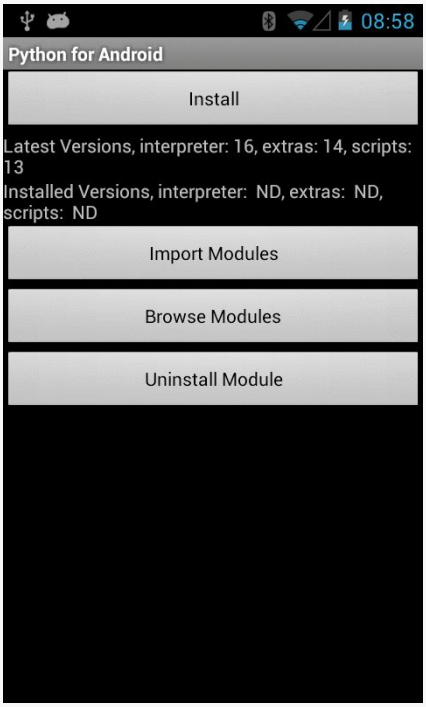


Figure 710: Downloading the Python SL4A interpreter, continued

Click the Install button, and SL4A will download and install the interpreter's component parts:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 711: Downloading the Python SL4A interpreter

This may take one or several downloads, depending on the interpreter. When done, and after a few progress dialogs' worth of unpacking, the interpreter will appear in the list of interpreters:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

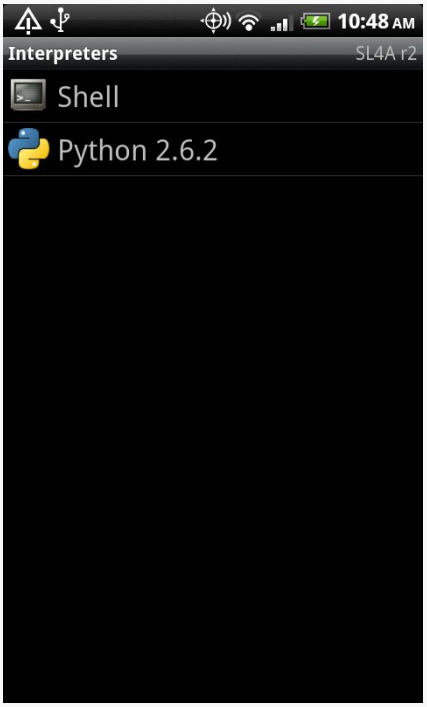


Figure 712: The updated list of installed SL4A interpreters

Note that the interpreters will be installed on your device's "external storage" (typically some flavor of SD card), due to their size. You will find an `SL4A/` directory on that card with the interpreters and scripts.

Running Supplied Scripts

Back on the Scripts activity (e.g., what you see when you launch SL4A from the launcher), you will be presented with a list of the available scripts. Initially, these will be ones that shipped with the interpreters, as examples for how to write SL4A scripts in that language:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

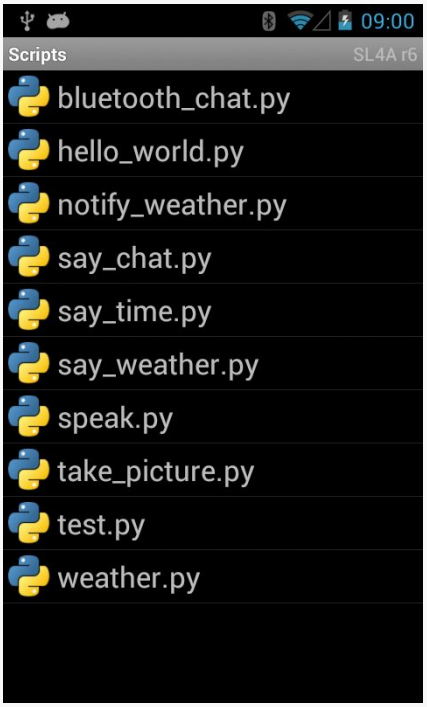


Figure 713: The list of SL4A scripts

Tapping on any of these scripts will bring up a "quick actions" balloon:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

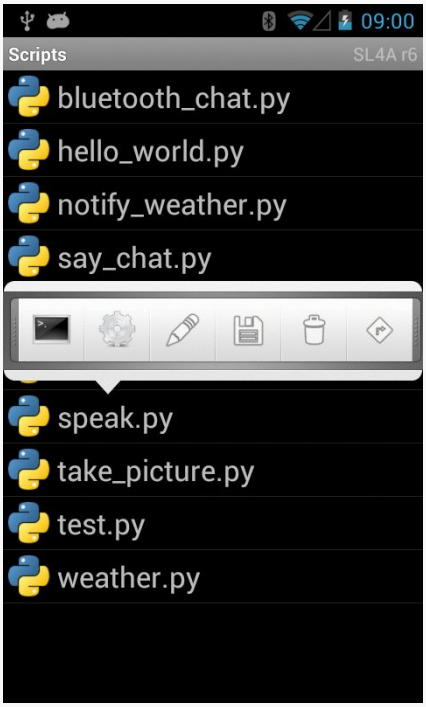


Figure 714: Quick actions for the speak.py script

Click the little shell icon to run it, showing its terminal output along the way:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

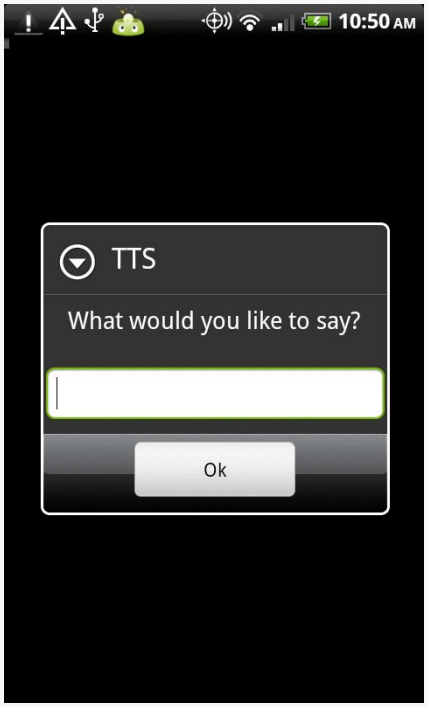


Figure 715: The visual results of running the speak.py SL4A script

Writing SL4A Scripts

While the scripts supplied with the interpreters are... entertaining, they only scratch the surface of what an SL4A script can accomplish. Of course, to go beyond what is there, you will need to start writing some scripts.

Editing Options

Since scripts are stored on your SD card (or whatever the "external storage" is for your device), you can create scripts using some other computer — one with fancy things like "mice" and "ergonomic keyboards" — and transfer it over via USB, like you would transfer over an MP3 file. This eases typing, but it will make for an awkward development cycle, since your computer and the Android device cannot both have access to the SD card simultaneously. The mount/unmount process may get a bit annoying. On the other hand, this is a great way to transfer over a script you obtained from somebody else.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Another option is to edit your scripts on the device. SL4A has a built in script editor designed for this purpose. Of course, the screen may be a bit small and the keyboard may be a bit... soft, but this is a great answer for small scripts.

To add a new script, from the Scripts activity, choose Add from the option menu. This will bring up a roster of available scripting languages and other items (e.g., add a folder):

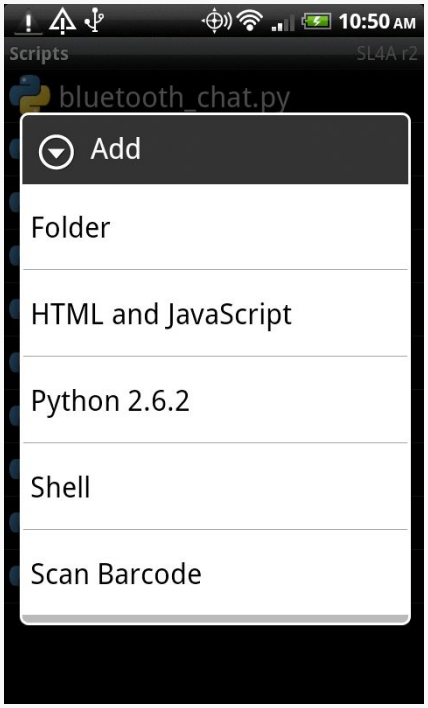


Figure 716: The add-script language selection dialog

(the "Scan Barcode" option gives you an easy route to install a third-party script, one encoded in a QR code)

Tap the language you want, and you will be taken into the script editor:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 717: The script editor

The field at the top is for the script name, and the large text area at the bottom is for the script itself. A file extension and boilerplate code will be supplied for you automatically.

In fact, that boilerplate code is rather important, as you will see momentarily.

To edit an existing script, long-tap on the script in the list and choose Edit from the context menu.

To save your changes to a new or existing script, choose the Save option from the script editor option menu. You can also "Save and Run" to test the script immediately.

Calling Into Android

In the real world, Perl knows nothing about Android. Neither does Python, BeanShell, or most of the other scripting languages available for SL4A. This would be rather limiting, as most of what you would want a script to do will have to deal with the device to some level: collect input, get a location, say some text using speech synthesis, dial the phone, etc.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Fortunately, SL4A has a solution, one of those "so crazy, it just might work" sorts of solutions: SL4A has a built-in RPC server. While implementing a server on a smartphone is not something one ordinarily does, it provides an ingenious bridge from the scripting language to the device itself.

Each scripting language is given a local object proxy that works with the RPC server. For example, here is a Python script that speaks the current time:

Figure 718: The script editor, showing the say_time.py script

The `import android` and `droid=android.Android()` statements establish a connection between the Python interpreter and the SL4A RPC server. From that point, the `droid` object is available for use to access Android capabilities — in this case, speaking a message.

Python does not strictly realize that it is accessing local functionality. It simply makes RPC calls, ones that just so happen to be fulfilled on the device rather than via some remote RPC server accessed over the Internet.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Browsing the API

Therefore, SL4A effectively exposes an API to each of its scripting languages, via this RPC bridge. While the API is not huge, it accomplishes a lot and is ever-growing.

If you are editing scripts on the device, you can browse the API by choosing the API Browser option menu from the script editor. This brings up a list of available methods on your RPC proxy (e.g., `droid`) that you can call:

Figure 719: The script editor's API browser

Tapping on any item in the list will "unfold" it to provide more details, such as the parameter list. Long-tapping on an item brings up a context menu where you can:

1. insert a template call to the method into your script at the cursor position
2. "prompt" you for the parameter values for the method, then insert the completed method call into your script

It is also possible to browse the API in a regular Web browser, if you are developing scripts off-device.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Running SL4A Scripts

Scripts are only useful if you run them, of course. We have seen two options for running scripts: tapping on them in the scripts list, or choosing "Save & Run" from the script editor. Those are not your only options, however.

Background

If you long-tap on a script in the script list, you will see a context menu option to "Start in Background". As the name suggests, this kicks off the script in the background. Rather than seeing the terminal window for the script, the script just runs. A notification will appear in the status bar, with the SL4A icon, indicating that the RPC server is in operation and that script(s) may be running.

Shortcuts

Rather than have to open up SL4A every time, you can set up shortcuts on your home screen to run individual scripts.

Android 1.x/2.x

Just long-tap on the home screen background and choose Shortcuts from the context menu, then Scripts from the available shortcuts. This brings up the scripts list, but this time, when you choose a script, you are presented with a quick actions balloon for how to start it: in a terminal or in the background:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

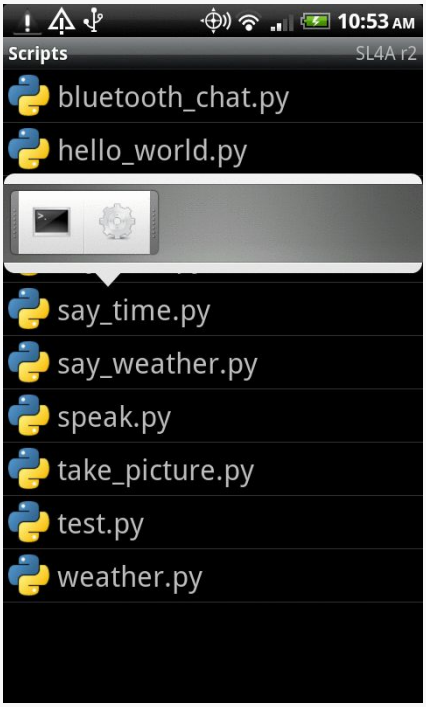


Figure 720: Configuring an SL4A shortcut

Choose one, and at this point, a shortcut, with the interpreter's icon and the name of the script, will appear on your home screen. Tapping it runs the script.

Android 3.0+

Go to where you install home screen widgets (e.g., "Widgets" tab in launcher), and you should see a "Scripts" entry:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

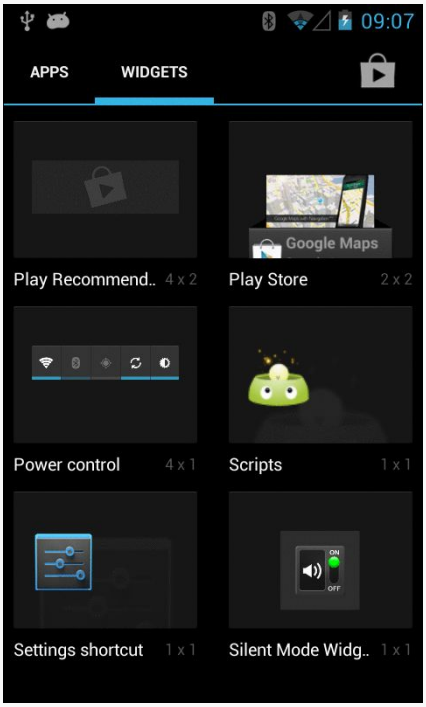


Figure 721: SL4A Shortcuts Option on Nexus S, Android 4.1

Add that to your home screen, the same as you would any other app widget (e.g., long press, then drag to the desired spot). That, in turn, will bring up the list of available scripts. Tapping on a script, as with Android 1.x/2.x, brings up a quick actions balloon for how to start it: in a terminal or in the background (see screenshot in previous section). This will then set up your shortcut on your home screen, so tapping it will launch your chosen script.

Other Alternatives

Users of Locale — an application designed to trigger events at certain times or when you get to certain locations — can trigger SL4A scripts in addition to invoking standard built-in tools.

In addition, there is preliminary support in SL4A for packaging scripts as APK files for wider distribution.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Potential Issues

As the SL4A Web site indicates, SL4A is "alpha-quality". It is not without warts. How much those warts are an issue for you, in terms of crafting and running utility scripts, is up to you.

Security… From Scripts

SL4A itself holds a long list of Android permissions, including:

1. The ability to read your contact data
2. The ability to call phone numbers and place SMS messages
3. Access to your location
4. Access to your received SMS/MMS messages
5. Bluetooth access
6. Internet access
7. The ability to write to the SD card
8. The ability to record audio and take pictures
9. The ability to keep your device awake
10. The ability to retrieve the list of running applications and restart other applications
11. And so on

Hence, its scripts — via the RPC-based API — can perform all of those actions. For example, a script you download from a third party could read all your contacts and send that information to a spammer. Hence, you should only run scripts that you trust, since SL4A effectively "wires open" many aspects of Android's standard security protections.

Security… From Other Apps

Originally, the on-device Web service supplying the RPC-based API was wide open. Any program that could find the port could connect to that Web service and invoke operations. That would not necessarily be all that bad… except that the Web service runs in its own process with its own permissions, and it may have permissions that other applications lack (e.g., right to access the Internet or to read contacts). Given that, malware could use SL4A to do things that it, by itself, could not do, allowing it to sneak onto more devices.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

SL4A now uses a token-based authentication mechanism for using the Web service, to help close this loophole. In principle, only SL4A scripts should be able to use the RPC server.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

JVM Scripting Languages

The Java virtual machine (JVM) is a remarkably flexible engine. While it was originally developed purely for Java, it has spawned its own family of languages, just as Microsoft's CIL supports multiple languages for the Windows platform. Some languages targeting the JVM as a runtime will work on Android, since the regular Java VM and Android's Dalvik VM are so similar.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Some of the sample code demonstrates JUnit test cases, so reading the chapter on unit testing may be useful.

Languages on Languages

Except for the handful of early language interpreters and compilers hand-constructed in machine code, every programming language is built atop earlier ones. C and C++ are built atop assembly language. Many other languages, such as Java itself, are built atop C/C++.

Hence, it should not come as much of a surprise that an environment as popular as Java has spawned another generation of languages whose implementations are in Java.

There are a few flavors of these languages. Some, like Scala and Clojure, are compiled languages whose compilers created JVM bytecodes, no different than would a Java compiler. These do not strictly qualify as a "scripting language", however, since they typically compile their source code to bytecode ahead of time.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Some Java-based scripting languages use fairly simple interpreters. These interpreters convert scripting code into parsed representations (frequently so-called "abstract syntax trees", or ASTs), then execute the scripts from their parsed forms. Most scripting languages at least start here, and some, like BeanShell, stick with this implementation.

Other scripting languages try to bridge the gap between a purely interpreted language and a compiled one like Scala or Clojure. These languages turn the parsed scripting code into JVM bytecode, effectively implementing their own just-in-time compiler (JIT). Since many Java runtimes themselves have a JIT to turn bytecode into machine code ("opcode"), languages with their own JIT can significantly outperform their purely-interpreted counterparts. JRuby and Rhino are two languages that have taken this approach.

A Brief History of JVM Scripting

Back in the beginning, the only way to write for the JVM was in Java itself. However, since writing language interpreters is a common pastime, it did not take long for people to start implementing interpreters in Java. These had their niche audiences, but there was only modest interest in the early days — interpreters made Java applets too large to download, for example.

Things got a bit more interesting in 1999, when IBM released the Bean Scripting Framework (BSF). This offered a uniform API for scripting engines, meaning that a hosting Java application could write to the BSF API, then plug in arbitrary interpreters at runtime. It was even possible, with a bit of extra work, to allow new interpreters to be downloaded and used on demand, rather than having to be pre-installed with the application. BSF also standardized how to inject Java objects into the scripting engines themselves, for access by the scripts. This allowed scripts to work with the host application's objects, such as allowing scripts to manipulate the contents of the jEdit text editor.

This spurred interest in scripting. In addition to some IBM languages (e.g., NetREXX) supporting BSF natively, other languages, like BeanShell, created BSF adapters to allow their languages to participate in the BSF space. On the consumer side, various Web frameworks started supporting BSF scripting for dynamic Web content generation, and so forth.

Interest was high enough that Apache took over stewardship of BSF in 2003. Shortly thereafter, Sun and others started work on JSR–223, which added the `javax.script`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

framework to Java 6. The `javax.script` framework advanced the BSF concept and standardized it as part of Java itself.

At this point, most JVM scripting languages that are currently maintained support `javax.script` integration, and may also support integration with the older BSF API as well.

Android does not include `javax.script` as part of its subset of the Java SE class library from the Apache Harmony project. This certainly does not preclude integrating scripting languages into Android applications, but it does raise the degree of difficulty a bit.

Limitations

Of course, JVM scripting languages do not necessarily work on Android without issue. There may be some work to get a JVM language going on Android, above and beyond the challenges for scripting languages in general on Android.

Android SDK Limits

Android is not Java SE, or Java ME, or even Java EE. While Android has many standard Java classes, it does not have a class library that matches any traditional pattern. As such, languages built assuming Java SE, for example, may have some dependency issues.

For languages where you have access to the source code, removing these dependencies may be relatively straightforward, particularly if they are ancillary to the operation of the language itself. For example, the language may come with miniature Swing IDEs, support for scripted servlets, or other capabilities that are not particularly relevant on Android and can be excised from the source code.

Wrong Bytecode

Android runs Dalvik bytecode, not Java bytecode. The conversion from Java bytecode to Dalvik bytecode happens at compile time. However, the conversion tool is rather finicky — it wants bytecode from Sun/Oracle's Java 1.5 or 1.6, nothing else. This can cause some problems:

1. You may encounter a JAR that is old enough to have been compiled with Java 1.4.2

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. You may encounter JARs compiled using other compilers, such as the GNU Compiler for Java (GCJ), common on Linux distributions
3. Java 7 has bytecode differences from Java 6; users of Java 7 need to compile their Java classes to Java 6 bytecode
4. Languages that have their own JIT compilers will have problems, because their JIT compilers will be generating Java bytecodes, not Dalvik bytecodes, meaning that the JIT facility needs to be rewritten or disabled

Again, if you have the source code, recompiling on an Android-friendly Java compiler should be a simple process.

Age

The heyday of some JVM languages is in the past. As such, you may find that support for some languages will be limited, simply because few people are still interested in them. Finding people interested in those languages on Android — the cross-section of two niches – may be even more of a problem.

SL4A and JVM Languages

SL4A supports three JVM languages today:

1. BeanShell
2. JRuby
3. Rhino (JavaScript)

You can use those within your SL4A environment no different than you can any other scripting language (e.g., Perl, Python, PHP). Hence, if what you are looking for is to create your own personal scripts, or writing small applications, SL4A saves you a lot of hassle. If there is a JVM scripting language you like but is not supported by SL4A, adding support for new interpreters within SL4A is fairly straightforward, though the APIs may change as SL4A is undergoing a fairly frequent set of revisions.

Embedding JVM Languages

While SL4A will drive end users towards writing their own scripts or miniature applications using JVM languages, another use of these languages is for embedding in a full Android application. Scripting may accelerate development, if the developers are more comfortable with the scripted language than with Java. Also, if

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the scripts are able to be modified or expanded by users, an ecosystem may emerge for user-contributed scripts.

Architecture for Embedding

Embedding a scripting language is not something to be undertaken lightly, even on a desktop or server application. Mobile devices running Android will have similar issues.

Asynchronous

One potential problem is that a script may take too long to execute. Android's architecture assume that work triggered by buttons, menus, and the like will either happen very quickly or will be done on background threads. Particularly for user-generated scripts, the script execution time is unknowable in advance — it might be a few milliseconds, or it might be several seconds. Hence, any implementation of a scripting extension for an Android application needs to consider executing all scripts in a background thread. This, of course, raises its own challenges for reflecting those scripts' results on-screen, since GUI updates cannot be done on a background thread.

Security

Scripts in Android inherit the security restrictions of the process that runs the script. If an application has the right to access the Internet, so will any scripts run in that application's process. If an application has the right to read the user's contacts, so will any scripts run in that application's process. And so on. If the scripts in question are created by the application's authors, this is not a big deal — the rest of the application has those same permissions, after all. But, if the application supports user-authored scripts, it raises the potential of malware hijacking the application to do things that the malware itself would otherwise lack the rights to do.

Inside the InterpreterService

One way to solve both of those problems is to isolate the scripting language in a self-contained low-permission APK — "sandboxing" the interpreter so the scripts it executes are less able to cause harm. This APK could also arrange to have the interpreter execute its scripts on a background thread. An even better implementation would allow the embedding application to decide whether or not

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the “sandbox” is important — applications with a controlled source of scripts may not need the extra security or the implementation headaches it causes.

With that in mind, let us take a look at the JVM/InterpreterService sample project, one possible implementation of the strategy described above.

The Interpreter Interface

The InterpreterService can support an arbitrary number of interpreters, via a common interface. This interface provides a simplified API for having an interpreter execute a script and return a result:

```
package com.commonsware.abj.interp;

import android.os.Bundle;

public interface I_Interpreter {
  Bundle executeScript(Bundle input);
}
```

As you can see, it is *very* simplified, offering just a single executeScript() method. That method accepts a Bundle (a key-value store akin to a Java HashMap) as a parameter — that Bundle will need to contain the script and any other objects needed to execute the script.

The interpreter will return another Bundle from executeScript(), containing whatever data it wants the script’s requester to have access to.

For example, here is the implementation of EchoInterpreter, which just returns the same Bundle that was passed in:

```
package com.commonsware.abj.interp;

import android.os.Bundle;

public class EchoInterpreter implements I_Interpreter {
  public Bundle executeScript(Bundle input) {
    return(input);
  }
}
```

A somewhat more elaborate sample is the SQLiteInterpreter:

```
package com.commonsware.abj.interp;

import android.database.Cursor;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;

public class SQLiteInterpreter implements I_Interpreter {
  public Bundle executeScript(Bundle input) {
    Bundle result=new Bundle(input);
    String script=input.getString(InterpreterService.SCRIPT);

    if (script!=null) {
      SQLiteDatabase db=SQLiteDatabase.create(null);
      Cursor c=db.rawQuery(script, null);

      c.moveToFirst();

      for (int i=0;i<c.getColumnCount();i++) {
        result.putString(c.getColumnName(i), c.getString(i));
      }

      c.close();
      db.close();
    }

    return(result);
  }
}
```

This class accepts a script, in the form of a SQLite database query. It extracts the script from the `Bundle`, using a pre-defined key (`InterpreterService.SCRIPT`). Assuming there is such a script, it creates an empty in-memory database and executes the SQLite query against that database.

The results come back in the form of a `Cursor` — itself a key-value store. `SQLiteInterpreter` takes those results and pours them into a `Bundle` to be returned.

The `Bundle` being returned starts from a copy of the input `Bundle`, so the script requester can embed in the input `Bundle` any identifiers it needs to determine how to handle the results from executing this script.

`SQLiteInterpreter` is not terribly flexible, but you can use it for simple numeric and string calculations, such as the following script:

```
SELECT 1+2 AS result, 'foo' AS other_result, 3*8 AS third_result;
```

This would return a `Bundle` containing a key of `result` with a value of `3`, a key of `other_result` with a value of `foo`, and a key of `third_result` with a value of `24`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Of course, it would be nice to support more compelling interpreters, and we will examine a pair of those later in this chapter.

Loading Interpreters and Executing Scripts

Of course, having a nice clean interface to the interpreters does nothing in terms of actually executing them on a background thread, let alone sandboxing them. The `InterpreterService` class itself handles that.

`InterpreterService` is an `IntentService`, which automatically routes incoming `Intent` objects (from calls to `startService()`) to a background thread via a call to `onHandleIntent()`. `IntentService` will queue up `Intent` objects if needed, and `IntentService` even automatically shuts down if there is no more work to be done.

Here is the implementation of `onHandleIntent()` from `InterpreterService`:

```
  @Override
  protected void onHandleIntent(Intent intent) {
    String action=intent.getAction();
    I_Interpreter interpreter=interpreters.get(action);

    if (interpreter==null) {
      try {
        interpreter=(I_Interpreter)Class.forName(action).newInstance();
        interpreters.put(action, interpreter);
      }
      catch (Throwable t) {
        Log.e("InterpreterService", "Error creating interpreter", t);
      }
    }

    if (interpreter==null) {
      failure(intent, "Could not create interpreter: "+intent.getAction());
    }
    else {
      try {
        success(intent,
interpreter.executeScript(intent.getBundleExtra(BUNDLE)));
      }
      catch (Throwable t) {
        Log.e("InterpreterService", "Error executing script", t);

        try {
          failure(intent, t);
        }
        catch (Throwable t2) {
          Log.e("InterpreterService",
                "Error returning exception to client",
                t2);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        }
      }
    }
  }
```

We keep a cache of interpreters, since initializing their engines may take some time. That cache is keyed by the interpreter's class name, and that key comes in to the service by way of the action on the `Intent` that was used to start the service. In other words, the script requester tells us, by way of the `Intent` used in `startService()`, which interpreter to use.

Those interpreters are created using reflection. This way, `InterpreterService` has no compile-time knowledge of any given interpreter class. Interpreters can come and go, but `InterpreterService` remains the same.

Assuming an interpreter was found (either cached or newly created), we have it execute the script, with the input `Bundle` coming from an "extra" on the `Intent`. Methods named `success()` and `failure()` are then responsible for getting the results to the script requester... as will be seen in the next section.

Delivering Results

Script requesters can get the results of the script back — in the form of the interpreter's output `Bundle` — in one of two ways.

One option is a private broadcast `Intent`. This is a broadcast `Intent` where the broadcast is limited to be delivered only to a specific package, not to any potential broadcast receiver on the device.

The other option is to supply a `PendingIntent` that will be sent with the results. This could be used by an `Activity` and `createPendingIntent()` to have control routed to its `onActivityResult()` method. Or, an arbitrary `PendingIntent` could be created, to start another activity, for example.

The implementations of `success()` and `failure()` in `InterpreterService` simply build up an `Intent` containing the results to be delivered:

```
  private void success(Intent intent, Bundle result) {
    Intent data=new Intent();

    data.putExtras(result);
    data.putExtra(RESULT_CODE, SUCCESS);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    send(intent, data);
  }

  private void failure(Intent intent, String message) {
    Intent data=new Intent();

    data.putExtra(ERROR, message);
    data.putExtra(RESULT_CODE, FAILURE);

    send(intent, data);
  }

  private void failure(Intent intent, Throwable t) {
    Intent data=new Intent();

    data.putExtra(ERROR, t.getMessage());
    data.putExtra(TRACE, getStackTrace(t));
    data.putExtra(RESULT_CODE, FAILURE);

    send(intent, data);
  }
```

These, in turn, delegate the actual sending logic to a `send()` method that delivers the result `Intent` via a private broadcast or a `PendingIntent`, as indicated by the script requester:

```
  private void send(Intent intent, Intent data) {
    String broadcast=intent.getStringExtra(BROADCAST_ACTION);

    if (broadcast==null) {
      PendingIntent pi=(PendingIntent)intent.getParcelableExtra(PENDING_RESULT);

      if (pi!=null) {
        try {
          pi.send(this, Activity.RESULT_OK, data);
        }
        catch (PendingIntent.CanceledException e) {
          // no-op -- client must be gone
        }
      }
    }
    else {
      data.setPackage(intent.getStringExtra(BROADCAST_PACKAGE));
      data.setAction(broadcast);

      sendBroadcast(data);
    }
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Packaging the InterpreterService

There are three steps for integrating `InterpreterService` into an application.

First, you need to decide what APK the `InterpreterService` goes in – the main one for the application (no sandbox) or a separate low-permission one (sandbox).

Second, you need to decide what interpreters you wish to support, writing `I_Interpreter` implementations and getting the interpreters' JARs into the project's `libs/` directory.

Third, you need to add the source code for `InterpreterService` along with a suitable `<service>` entry in `AndroidManifest.xml`. This entry will need to support `<intent-filter>` elements for each scripting language you are supporting, such as:

```
    <service
      android:name=".InterpreterService"
      android:exported="false">
      <intent-filter>
        <action android:name="com.commonsware.abj.interp.EchoInterpreter"/>
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.abj.interp.SQLiteInterpreter"/>
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.abj.interp.BshInterpreter"/>
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.abj.interp.RhinoInterpreter"/>
      </intent-filter>
    </service>
```

From there, it is a matter of adding in appropriate `startService()` calls to your application wherever you want to execute a script, and processing the results you get back.

Using the InterpreterService

To use the `InterpreterService`, you need to first determine which `I_Interpreter` engine you are using, as that forms the action for the `Intent` to be used with the `InterpreterService`. Create an `Intent` with that action, then add in an `InterpreterService.BUNDLE` extra for the script and other data to be supplied to the interpreter. Also, you can add an `InterpreterService.BROADCAST_ACTION`, to be used by `InterpreterService` to send results back to you via a broadcast `Intent`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Finally, call `startService()` on the `Intent`, and the results will be delivered to you asynchronously.

For example, here is a test method from the `EchoInterpreterTests` test case:

```
package com.commonsware.abj.interp;

import android.os.Bundle;

public class EchoInterpreterTests extends InterpreterTestCase {
  protected String getInterpreterName() {
    return("com.commonsware.abj.interp.EchoInterpreter");
  }

  public void testNoInput() {
    Bundle results=execServiceTest(new Bundle());

    assertNotNull(results);
    assert (results.size() == 0);
  }

  public void testWithSomeInputJustForGrins() {
    Bundle input=new Bundle();

    input.putString("this", "is a value");

    Bundle results=execServiceTest(input);

    assertNotNull(results);
    assertEquals(results.getString("this"), "is a value");
  }
}
```

The echo "interpreter" simply echoes the input `Bundle` into the output. The `execServiceTest()` method is inherited from the `InterpreterTestCase` base class:

```
  protected Bundle execServiceTest(Bundle input) {
    Intent i=new Intent(getInterpreterName());

    i.putExtra(InterpreterService.BUNDLE, input);
    i.putExtra(InterpreterService.BROADCAST_ACTION, ACTION);

    getContext().startService(i);

    try {
      latch.await(5000, TimeUnit.MILLISECONDS);
    }
    catch (InterruptedException e) {
      // just keep rollin'
    }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    return(results);
  }
```

The `execServiceTest()` method uses a `CountDownLatch` to wait on the interpreter to do its work before proceeding (or 5000 milliseconds, whichever comes first). The broadcast `Intent` containing the results, registered to watch for `com.commonsware.abj.interp.InterpreterTestCase` broadcasts, stuffs the output `Bundle` in a results data member and drops the latch, allowing the main test thread to continue.

BeanShell on Android

What if Java itself were a scripting language? What if you could just execute a snippet of Java code, outside of any class or method? What if you could still import classes, call static methods on classes, create new objects, as well?

That was what BeanShell offered, back in its heyday. And, since BeanShell does not use sophisticated tricks with its interpreter – like JIT compilation of scripting code — BeanShell is fairly easy to integrate into Android.

What is BeanShell?

BeanShell is Java on Java.

With BeanShell, you can write scripts in loose Java syntax. Here, "loose" means:

1. In addition to writing classes, you can execute Java statements outside of classes, in a classic imperative or scripting style
2. Data types are optional for variables
3. Not every language feature is supported, particularly things like annotations that did not arrive until Java 1.5
4. Etc.

BeanShell was originally developed in the late 1990's by Pat Niemeyer. It enjoyed a fair amount of success, even being considered as a standard interpreter to ship with Java (JSR–274). However, shortly thereafter, BeanShell lost momentum, and it is no longer being actively maintained. That being said, it works quite nicely on Android... once a few minor packaging issues are taken care of.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Getting BeanShell Working on Android

BeanShell has two main problems when it comes to Android:

- The publicly-downloadable JAR was compiled for Java 1.4.2, and Android requires Java 5 or newer
- The source code includes various things, like a Swing-based GUI and a servlet, that have no real place in an Android app and require classes that Android lacks

Fortunately, with BeanShell being open source, it is easy enough to overcome these challenges. You could download the source into an Android library project, then remove the classes that are not necessary (e.g., the servlet), and use that library project in your main application. Or, you could use an Android project for creating a JAR file that was compiled against the Android class library, so you are certain everything is supported.

However, the easiest answer is to use SL4A's BeanShell JAR, since they have solved those problems already. The JAR can be found in the SL4A source code repository, though you will probably need to check out the project using Mercurial, since JARs cannot readily be downloaded from the Google Code Web site.

Integrating BeanShell

The BeanShell engine is found in the `bsh.Interpreter` class. Wrapping one of these in an `I_Interpreter` interface, for use with `InterpreterService`, is fairly simple:

```
package com.commonsware.abj.interp;

import android.os.Bundle;
import bsh.Interpreter;

public class BshInterpreter implements I_Interpreter {
  public Bundle executeScript(Bundle input) {
    Interpreter i=new Interpreter();
    Bundle output=new Bundle(input);
    String script=input.getString(InterpreterService.SCRIPT);

    if (script != null) {
      try {
        i.set(InterpreterService.BUNDLE, input);
        i.set(InterpreterService.RESULT, output);

        Object eval_result=i.eval(script);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        output.putString("result", eval_result.toString());
      }
      catch (Throwable t) {
        output.putString("error", t.getMessage());
      }
    }

    return(output);
  }
}
```

BeanShell interpreters are fairly inexpensive objects, so we create a fresh `Interpreter` for each script, so one script cannot somehow access results from prior scripts. After setting up the output `Bundle` and extracting the script from the input `Bundle`, we inject both `Bundle` objects into BeanShell itself, where they can be accessed like global variables, named `_bundle` and `_result`.

At this point, we evaluate the script, using the `eval()` method on the `Interpreter` object. If all goes well, we convert the object returned by the script into a `String` and tuck it into the output `Bundle`, alongside anything else the script may have put into the `Bundle`. If there is a problem, such as a syntax error in the script, we put the error message into the output `Bundle`.

So long as the `InterpreterService` has an `<intent-filter>` for the `com.commonsware.abj.interp.BshInterpreter` action, and so long as we have a BeanShell JAR in the project's `libs/` directory, `InterpreterService` is now capable of executing BeanShell scripts as needed.

With our inherited `execServiceTest()` method handling invoking the `InterpreterService` and waiting for responses, we can "simply" put our script as the `InterpreterService.SCRIPT` value in the input `Bundle`, and see what we get out. The first test script returns a simple value; the second test script directly calls methods on the output `Bundle` to return its results.

Rhino on Android

JavaScript arrived on the language scene hot on the heels of Java itself. The name was chosen for marketing purposes more so than for any technical reason. Java and JavaScript had little to do with one another, other than both adding interactivity to Web browsers. And while Java has largely faded from mainstream browser usage, JavaScript has become more and more of a force on the browser, and even now on Web servers.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

And, along the way, the Mozilla project put JavaScript on Java and gave us Rhino.

What is Rhino?

If BeanShell is Java in Java, Rhino is JavaScript in Java.

As part of Netscape's failed "Javagator" attempt to create a Web browser in Java, they created a JavaScript interpreter for Java, code-named Rhino after the cover of O'Reilly Media's JavaScript: The Definitive Guide. Eventually, Rhino was made available to the Mozilla Foundation, which has continued maintaining it. At the present time, Rhino implements JavaScript 1.7, so it does not support the latest and greatest JavaScript capabilities, but it is still fairly full-featured.

Interest in Rhino has ticked upwards, courtesy of interest in using JavaScript in places other than Web browsers, such as server-side frameworks. And, of course, it works nicely with Android.

Getting Rhino Working on Android

Similar to BeanShell, Rhino has a few minor source-level incompatibilities with Android. However, these can be readily pruned out, leaving you with a still-functional JavaScript interpreter. However, once again, it is easiest to use SL4A's Rhino JAR, since all that work is done for you.

Integrating Rhino

Putting an `I_Interpreter` facade on Rhino is incrementally more difficult than it is for BeanShell, but not by that much:

```
package com.commonsware.abj.interp;

import android.os.Bundle;
import org.mozilla.javascript.*;

public class RhinoInterpreter implements I_Interpreter {
  public Bundle executeScript(Bundle input) {
    String script=input.getString(InterpreterService.SCRIPT);
    Bundle output=new Bundle(input);

    if (script != null) {
      Context ctxt=Context.enter();

      try {
        ctxt.setOptimizationLevel(-1);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        Scriptable scope=ctxt.initStandardObjects();
        Object jsBundle=Context.javaToJS(input, scope);
        ScriptableObject.putProperty(scope, InterpreterService.BUNDLE,
                                     jsBundle);

        jsBundle=Context.javaToJS(output, scope);
        ScriptableObject.putProperty(scope, InterpreterService.RESULT,
                                     jsBundle);
        String result=
            Context.toString(ctxt.evaluateString(scope, script,
                                                 "<script>", 1, null));

        output.putString("result", result);
      }
      finally {
        Context.exit();
      }
    }

    return(output);
  }
}
```

As with `BshInterpreter`, `RhinoInterpreter` sets up the output `Bundle` and extracts the script from the input `Bundle`. Assuming there is a script, `RhinoInterpreter` then sets up a Rhino `Context` object, which is roughly analogous to the BeanShell `Interpreter` object. One key difference is that you need to clean up the `Context`, by calling a static `exit()` method on the `Context` class, whereas with a BeanShell `Interpreter`, you just let garbage collection deal with it.

Rhino has a JIT compiler, one that unfortunately will not work with Android, since it generates Java bytecode, not Dalvik bytecode. However, Rhino lets you turn that off, by calling `setOptimizationLevel()` on the `Context` object with a value of `-1` (meaning, in effect, disable all optimizations).

After that, we:

1. Create a language scope for our script and inject standard JavaScript global objects into that scope
2. Wrap our two `Bundle` objects with JavaScript proxies via calls to `javaToJS()`, then injecting those objects into the scope as

`_bundle` and `_result` via `putProperty()` calls

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. Execute the script via a call to `evaluateString()` on the `Context` object, converting the resulting object into a `String` and pouring it into the output `Bundle`

If our `InterpreterService` has an `<intent-filter>` for the `com.commonsware.abj.interp.RhinoInterpreter` action, and so long as we have a Rhino JAR in the project's `libs/` directory, `InterpreterService` can now invoke JavaScript.

Other JVM Scripting Languages

As mentioned previously, there are many languages that, themselves, are implemented in Java and can be ported to Android, with varying degrees of difficulty. Many of these languages are fairly esoteric. Some, like JRuby, have evolved to the point where they transcend a simple "scripting language" on Android.

However, there are two other languages worth mentioning, as they are fairly well-known in Java circles: Groovy and Jython.

Groovy

Groovy is perhaps the most popular Java-based language that does not have its roots in some other previous language (Java, JavaScript, Python, etc.). Designed in some respects to be a "better Java than Java", Groovy gives you access to Java classes while allowing you to write scripts with dynamic typing, closures, and so forth. Groovy has an extensive community, complete with a fair number of Groovy-specific libraries and frameworks, plus some books on the market.

At the time of this writing, it does not appear that Groovy has been successfully ported to work on Android, though.

Jython

Jython is an implementation of a Python language interpreter in Java. It has been around for quite some time, and gives you Python syntax with access to standard Java classes where needed. While the Jython community is not as well-organized as that of Groovy, there are plenty of books covering the use of Jython.

Jython's momentum has flagged a bit in recent months, in part due to Sun's waning interest in the technology and the departure of Sun employees from the project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

One attempt to get Jython working with Android has been shut down, with people steered towards SL4A. It is unclear if others will make subsequent attempts.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Other Tools

Special Creative Commons BY-NC-SA 4.0 License Edition

Advanced Emulator Capabilities

The Android emulator, at its core, is not that complex. Once you have one or more Android virtual devices (AVDs) defined, using them is a matter of launching the emulator and installing your app upon it. With Android Studio and Eclipse, those two steps can even be combined — the IDE will automatically start an emulator instance if one is needed.

However, there is much more to the Android emulator. This chapter will explore various advanced features of the emulator and how you can use them.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

x86 Images

Normally, the Android emulator emulates a device with an ARM-based CPU. That matches with most Android devices available to users today. However, most developers are developing on an x86-based development machine, not one powered by ARM. As a result, the normal Android emulator has to convert ARM instructions to x86 instructions before executing them, slowing down performance.

Some versions of the Android emulator, though, have an x86 version as well. Where available, these *can* run much more quickly than will their ARM counterparts on an x86 development machine.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The emphasis on *can* is that your development machine must have things set up properly first. Linux users need KVM, while Mac and Windows users need the "Intel Hardware Accelerated Execution Manager" (available from the SDK Manager). The latter must be manually installed once downloaded — please consult the Android tools documentation for details.

Also, this only works for certain CPU architectures, ones that support virtualization in hardware:

- Intel Virtualization Technology (VT, VT-x, vmx) extensions
- AMD Virtualization (AMD-V, SVM) extensions (Linux only)

Those virtualization extensions must also be enabled in your device's BIOS, and other OS-specific modifications may be required.

Android 4.0.3

An x86 image for Android 4.0.3 is available from your SDK Manager:

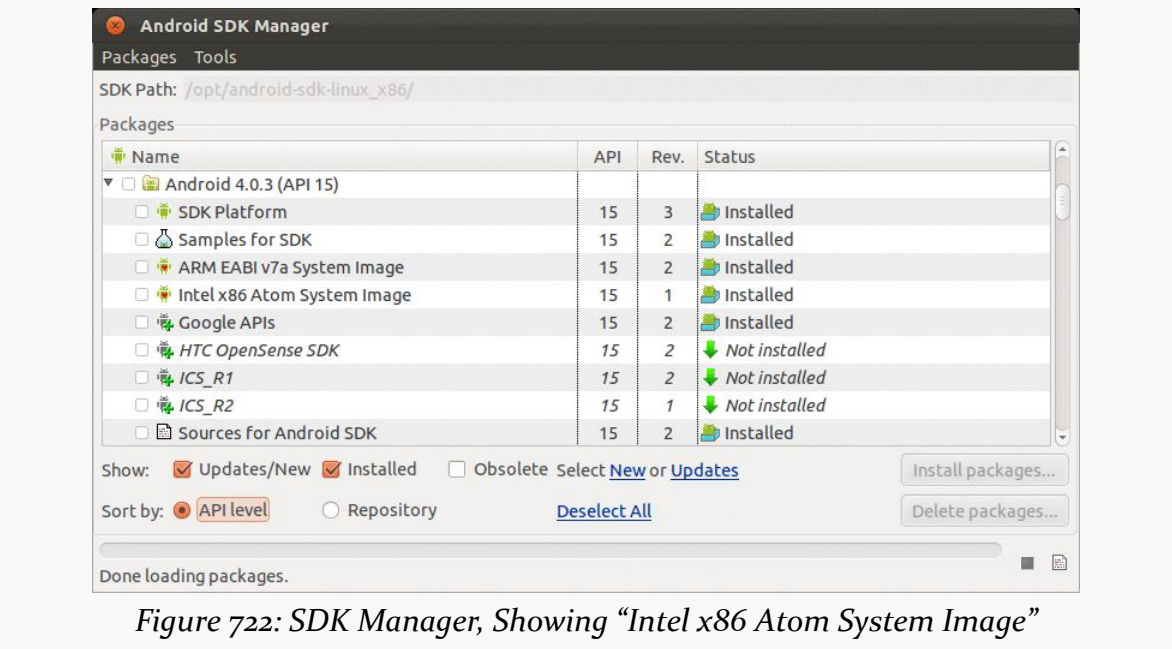


Figure 722: SDK Manager, Showing "Intel x86 Atom System Image"

When you download that, the next time you choose API Level 15 for an AVD, you will have an option of CPU architecture:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

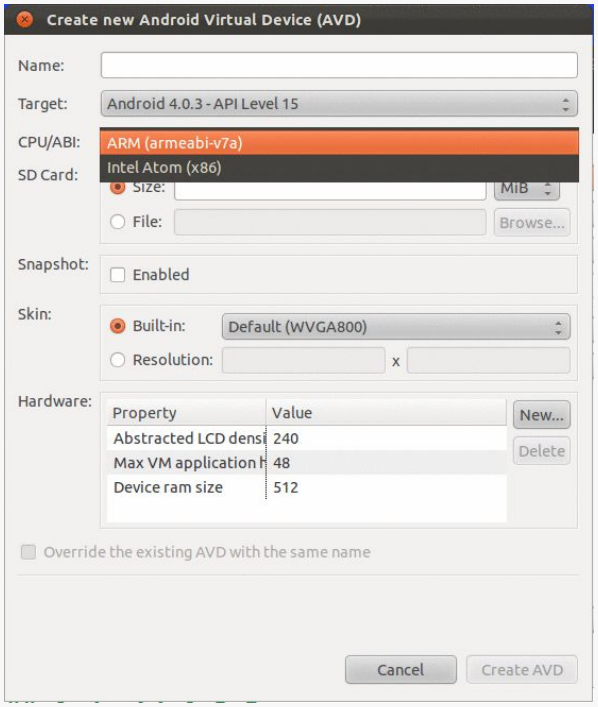


Figure 723: AVD Manager, Showing CPU/ABI Options

Note that this only works for the plain Android API Level 15 AVD, not the one containing Google Maps, which is only available for ARM at this time.

There are similar x86 images available for other newer Android versions, such as 4.1 and 4.2.

Android 2.3.3

An x86 image for Android 2.3.3 is also available from your SDK Manager, though with a slightly different entry:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

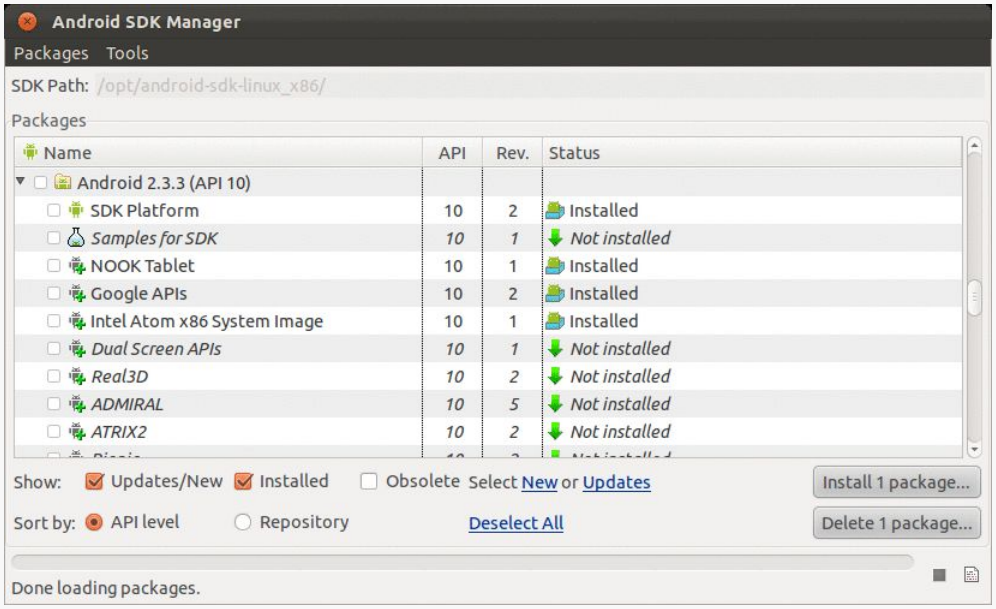


Figure 724: SDK Manager, Showing "Intel Atom x86 System Image"

This shows up as a separate target in your AVD Manager ("Intel Atom x86 System Image"), rather than a CPU/ABI value that you toggle.

Hardware Graphics Acceleration

The other way to speed up the emulator is to have it use the graphic card or GPU of your development machine to accelerate the graphics rendering of the emulator window. By default, the emulator will use software-based rendering, without the GPU, which is slow in general and worse when running an ARM-based image.

Whether this will work or not for you will depend in part upon your graphics drivers of your development machine.

There are two ways to configure GPU emulation, depending upon when you created your AVD.

Old AVDs

To try using GPU emulation, for an AVD (new or existing), click the "New..." button to the right of the list of hardware options in the AVD configuration editor:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

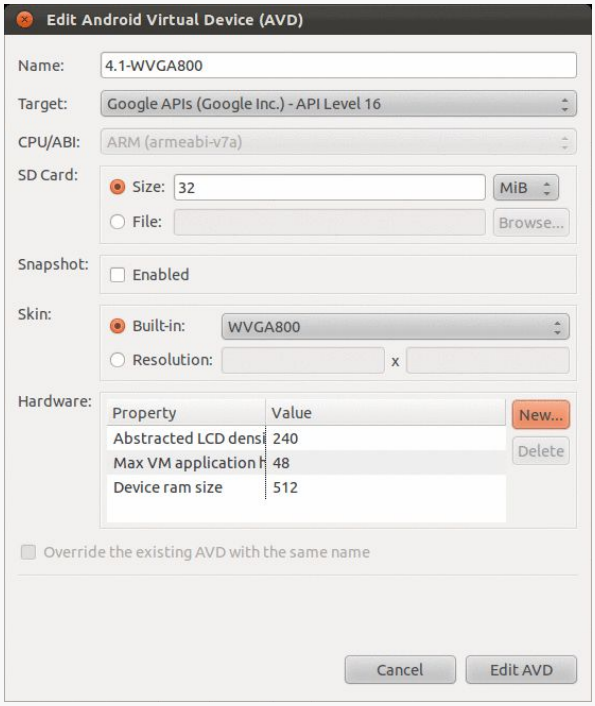


Figure 725: AVD Configuration Editor, With "New..." Button Focused

In the dialog that appears, choose "GPU Emulation" in the drop-down:

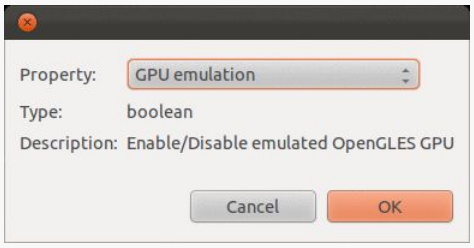


Figure 726: AVD Hardware Options Dialog

Then click OK, which will add "GPU Emulation" to the table:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

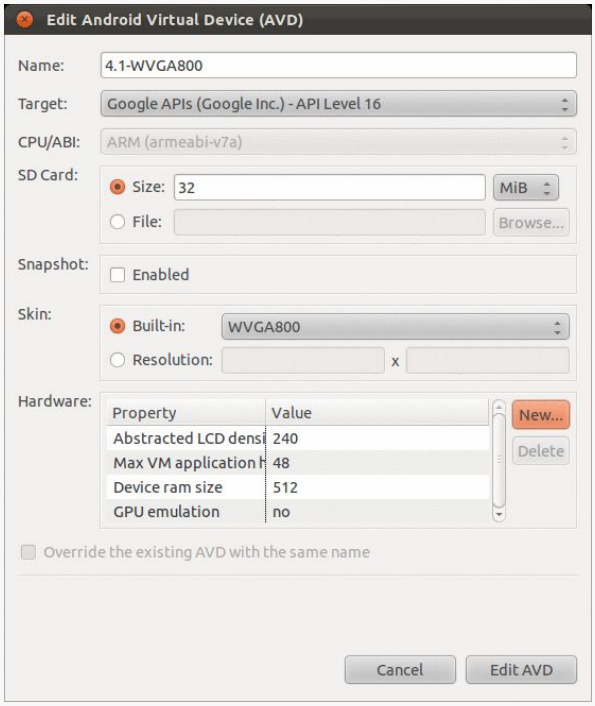


Figure 727: AVD Configuration Editor, Showing "GPU Emulation" Option

If it has "no" as the corresponding value — and it should by default — click on the "no" to display a drop-down where you can toggle it to "yes".

Also, you need to make sure that the "Enabled" checkbox in the Snapshot group box is unchecked.

New AVDs

On a newer AVD configuration editor, there is a "Use Host GPU" checkbox that you can check to try using GPU emulation:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

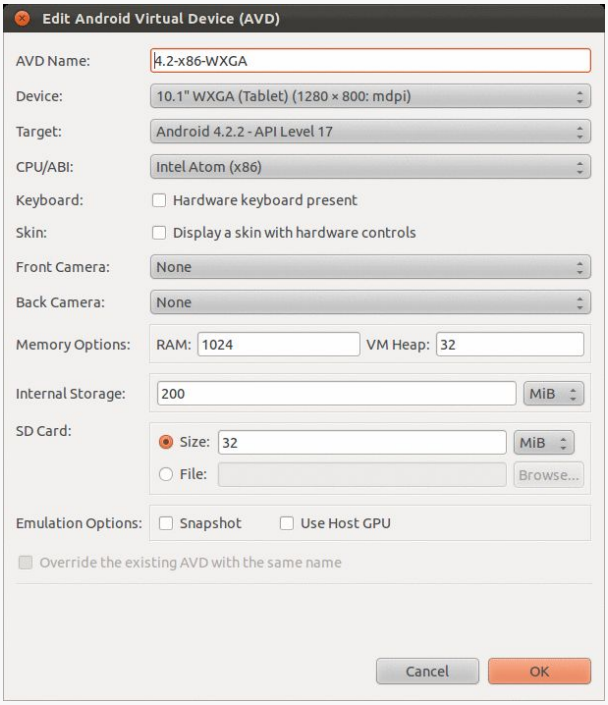


Figure 728: New AVD Configuration Editor, Showing "Use Host GPU" Option

Defining New Devices

Your Android development environment will have several device definitions ready for you to use. Some will come from Google. Others may come from device vendors, such as Amazon shipping device definitions for the Kindle Fire series. And you can define your own, to attempt to (roughly) model other devices beyond those already in your roster.

In the AVD Manager, on the Device Definitions tab, you can click the "New Device..." button on the right:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

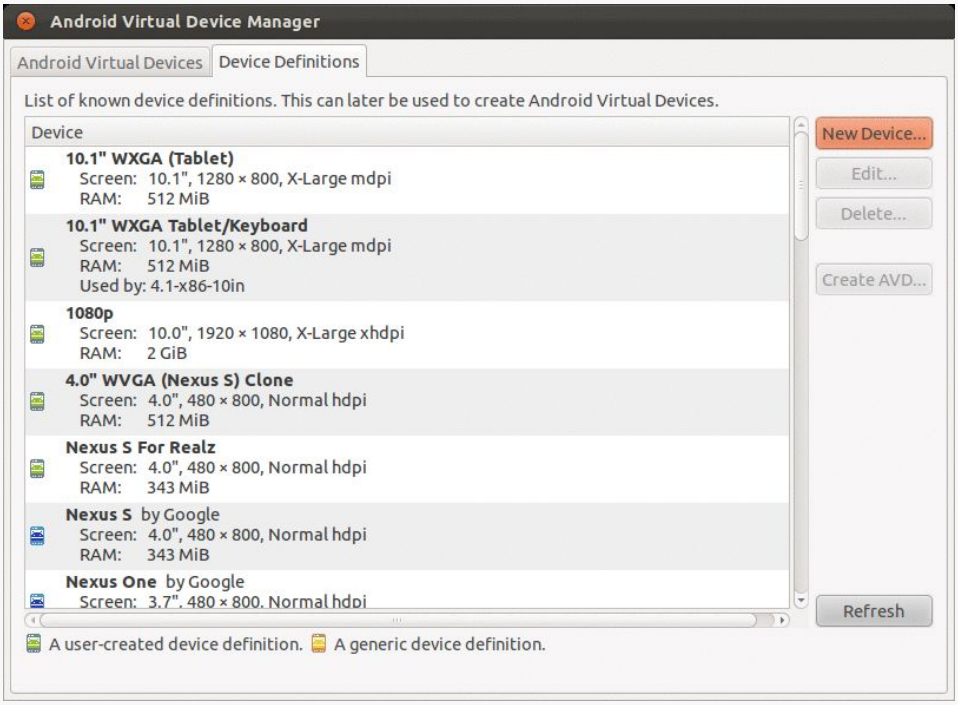


Figure 729: AVD Manager, Device Definitions Tab

That will bring up an empty device definition dialog:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

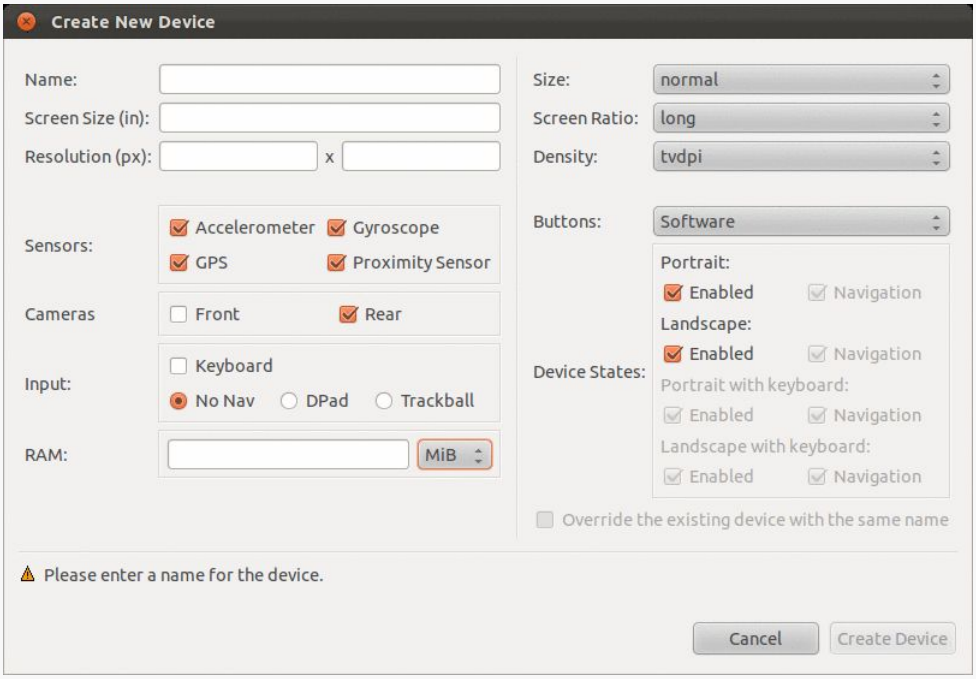


Figure 730: AVD Manager, Device Definitions Dialog

Alternatively, you can highlight an existing definition. If it is one you defined yourself, you will have an "Edit..." button to modify the definition. If it is a definition that came from Google or another vendor, that button will be labeled "Clone..." and will allow you to create a new definition cloned from the existing one. You can clone one of your own as well, by editing it and giving it a new name.

There are lots of things that you can configure for a device definition. The major ones are:

- The name of the definition, to control what it is called in the drop-down of available devices when you go to create an AVD
- The diagonal screen size of the device in inches, plus its resolution in pixels
- The screen size, ratio (long or not-long), and density buckets the emulator should use — note that these will be pre-populated based on your size and resolution values, but you can override them if needed
- What sensors the device should claim to have (e.g., accelerometer)
- What cameras the device should claim to have
- Whether the device has a keyboard for input and/or some sort of navigation pointer (e.g., a trackball)
- How much system RAM should be on the device

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- Whether or not the HOME, BACK, etc. buttons are implemented in hardware or in software
- What orientations are supported (and if you specified a keyboard, whether the keyboard is available in those orientations)

Keyboard Behavior

The Android emulator can emulate devices that have, or do not have, a physical keyboard. Most Android devices do not have a physical keyboard, and so the emulator is set up to behave the same. However, this means that typing on your development machine's keyboard will not work in `EditText` widgets and the like — you have to tap out what you want to type on the on-screen keyboard.

If you wish to switch your emulator to emulate a device with a physical keyboard – either "for realz" or just to simplify working with the emulator on your development machine — you can do so.

For older AVDs, click that "New..." button next to the list of hardware options, as described in the preceding section. Choose "Keyboard support" from the drop-down, click OK, and toggle the value for that hardware option to "yes".

For newer AVDs, check the "Hardware keyboard present" checkbox.

Headless Operation

Sometimes, you want an emulator without a GUI. Typically, this is used for continuous integration or some other server-based testing solution — you use the "headless" emulator to run tests, even on a machine that lacks any GUI capability.

To do this, you will need to run the emulator from the command-line. Run `emulator -no-window -avd ...`, where `...` is the name of your AVD (e.g., the value in the left column of the list of AVDs in the AVD Manager). To test this first in normal mode, run the command without the `-no-window` switch.

The simplest solution to get rid of the emulator instance is to `kill` its process.

There are many other command-line switches for the emulator that you may wish to investigate. While most of these have UI analogues in the AVD Manager, the switches would be necessary to replicate some of those for headless operation.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Using Lint

As C/C++ developers are well aware, `lint` is not merely something that collects in pockets and belly buttons.

`lint` is a long-standing C/C++ utility that points out issues in a code base that are not errors or warnings, but are still indicative of a likely flaw in the code. After all, what might be legal from a syntax standpoint may still be a bug when used.

The Android tools now have their own equivalent tool, Lint, integrated into Android Studio and Eclipse, for reporting similar sorts of issues with an Android project's Java code, resources, and manifest. You can also get Lint reports from the command line, such as via Gradle for Android, perhaps as part of integrating your builds into a continuous integration server.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

What It Is

Lint can be best described as "a pest, but a good pest".

Normally, what stops you from building your app are compiler errors: bad Java syntax, malformed XML resource files, and the like. At the command line, these stop an in-progress build and dump error messages to the console. In Android Studio, these are noted in a log and also by notations in the source code. In Eclipse, these result in red "X" notations on the files in the Package Explorer, and frequently result

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

in red sqiggle lines underneath the offending Java or XML when viewed in an editor. You also may get yellow squiggle lines for warnings — things the compiler will allow but the compiler thinks may be a problem.

However, there are many things that might be syntactically valid but are not a good idea from an Android standpoint. For example, if you specify a minimum SDK version of API Level 8, and you try using a class that only exists on API Level 11, that's a problem if you are not handling it correctly and avoiding this class on the older-yet-supported devices. Yet, if your build target is API Level 11 or higher, it is perfectly valid syntax and would compile just fine.

Lint is designed to encapsulate rules that transcend syntax, to add more errors and warnings that reflect good Android practices beyond simple validity.

When It Runs

Running Lint sometimes happens automatically (e.g., from your IDE) or sometimes happens manually. The following sections outline the various possibilities.

Android Studio

By default, in Android Studio, Lint will run when you save a file, giving you error (red) or warning (yellow) squiggles for things that run afoul of Lint rules:

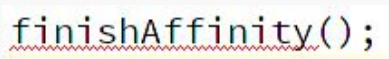


Figure 731: Android Studio Lint Error

You can manually invoke it via Analyze > Inspect Code... from the main menu, though this also seems to perform a bunch of other analysis that is not necessarily relevant for you as an Android developer, such as "J2ME issues".

Figure 732: Android Studio Inspection Results

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Eclipse

By default, in Eclipse, Lint will run when you save a file and when you export an APK (e.g., to distribute in production). You can also force a full Lint run in Eclipse at any point by clicking its toolbar button (looks like a green checkmark in a box), or by right-clicking over a project and choosing Android Tools > Run Lint from the context menu. In addition to giving you classic Eclipse error and warning markers in the files, there is also a "Lint Warnings" view showing a table of all the errors and warnings in one place:

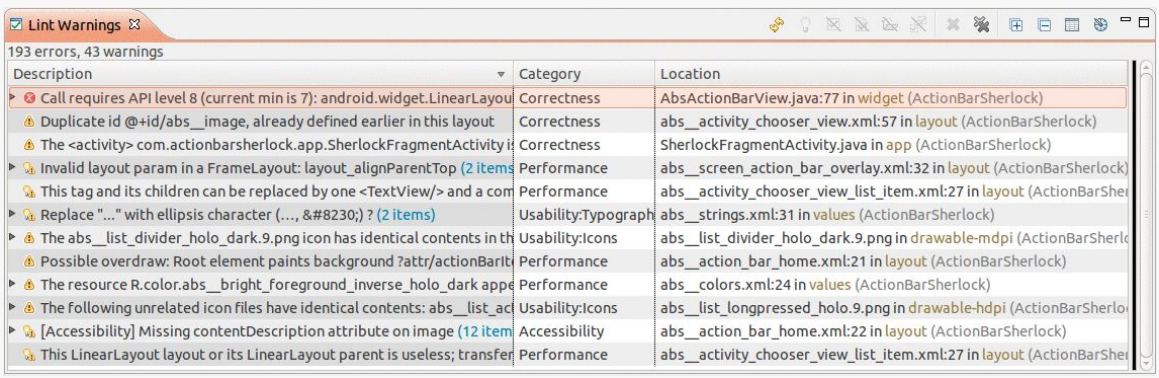


Figure 733: Eclipse Lint Warnings View

Command Line

To run Lint from the command line, just run `lint`, passing it the path to some directory. If the directory is an Android project directory, `lint` will dump the errors and warnings to the console. If the directory is not an Android project directory, `lint` will sweep all subdirectories to find any Android projects, then report those projects' errors and warnings.

```
$ lint .

Scanning .:
.....................................................................................
Scanning . (Phase 2): ..
res/drawable/eject.png: Warning: The resource R.drawable.eject appears to be
unused [UnusedResources]
res/values/strings.xml:3: Warning: The resource R.string.app_name appears to be
unused [UnusedResources]
    <string name="app_name">AudioDemo</string>
    ^
res/drawable-hdpi: Warning: Missing the following drawables in drawable-hdpi:
cw.png (found in drawable-mdpi) [IconDensities]
res: Warning: Missing density variation folders in res: drawable-xhdpi
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
[IconMissingDensityFolder]
res/layout/main.xml:13: Warning: [Accessibility] Missing contentDescription
attribute on image [ContentDescription]
    <ImageButton android:id="@+id/play"
    ^
res/layout/main.xml:35: Warning: [Accessibility] Missing contentDescription
attribute on image [ContentDescription]
    <ImageButton android:id="@+id/pause"
    ^
res/layout/main.xml:56: Warning: [Accessibility] Missing contentDescription
attribute on image [ContentDescription]
    <ImageButton android:id="@+id/stop"
    ^
res/layout/main.xml:21: Warning: [I18N] Hardcoded string "Play", should use
@string resource [HardcodedText]
      android:text="Play"
      ^
res/layout/main.xml:42: Warning: [I18N] Hardcoded string "Pause", should use
@string resource [HardcodedText]
      android:text="Pause"
      ^
res/layout/main.xml:63: Warning: [I18N] Hardcoded string "Stop", should use
@string resource [HardcodedText]
      android:text="Stop"
      ^
0 errors, 10 warnings
```

However, frequently it is more convenient as a developer to have the command-line `lint` generate an HTML report, instead of dumping everything just to the console. To do that, use the `--html` switch, passing a path to the report file to be generated. For local use, that is all you need. If you wish to host the report somewhere, also add the `--url` switch, indicating where the report will live on a Web server (e.g., your continuous integration server). For example, this command runs `lint` in the current working directory, generating a `/tmp/lint.html` file (plus a `/tmp/lint_files/` directory of images and CSS files), mapping the URLs to work on a specific base URL:

```
lint --html /tmp/lint.html --url .=http://misc.commonsware.com/lint
```

This report can be viewed in your Web browser to see what the output looks like.

You can also invoke Lint via `gradle lint` or a per-variant edition (e.g., `gradle lintRelease`). This will write results to an XML file in `build/outputs/` based upon product variant (e.g., `build/outputs/lint-results-release.xml` for a `gradle lintRelease` run). It will also emit an HTML file with the same base name in the same directory. These contain the same basic information as you get from the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

command-line output, with the XML in particular designed to be consumed by other tools, such as a continuous integration server.

What to Fix

In Android Studio, clicking on a red or yellow squiggle will pop up an adjacent "lightbulb" drop-down offering ways to fix the problem:

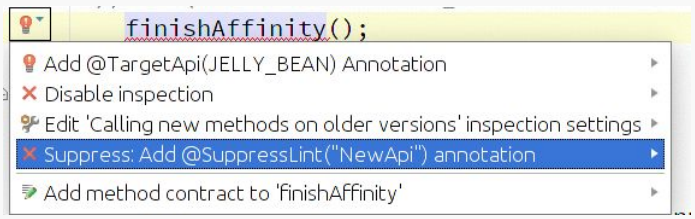


Figure 734: Android Studio Lint Fix Suggestions

Inside of Eclipse, some of the Lint warnings and errors come with "quick fixes", which you can bring up via `<Ctrl>-<1>`. For example:

- Errors related to accessing classes or methods higher than your `minSdkVersion` have "quick fixes" to add the `@TargetApi` annotation to the class or method containing your code
- Warnings related to hard-coded strings in layouts or the manifest have "quick fixes" to convert those strings into string resources

All warnings and errors will have "quick fixes" to suppress that warning or error in the future, by adding notations to the file to that effect.

What to Configure

You have some measure of control over Lint's behavior. The exact means of doing so varies significantly depending upon whether you are using Eclipse or running Lint from the command line.

Android Studio

In Android Studio, you can configure Lint's behavior via the Project Settings dialog, accessible via File > Settings:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

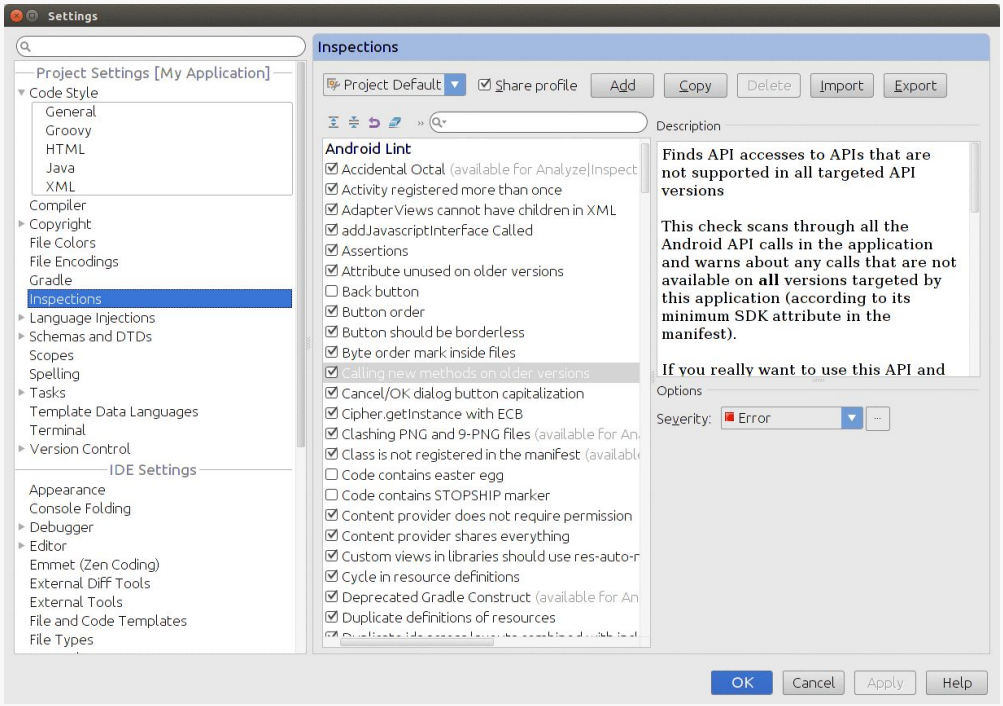


Figure 735: Android Studio Lint Error Checking Preferences

You can change some details about the specific checks that Lint makes:

- the severity of the issue, usually set to Warning or Error
- whether the specific issue should be ignored rather than executed

To do this, you may wish to create your own inspection profile, rather than modifying the stock "Project Default" profile. To do this, just click the "Copy" button in the Inspections page of the Settings dialog and supply a name for the new profile.

The above recipe changes the inspections for the individual project. To change them for new projects, go into File > Other Settings > Default Settings, and make your changes there.

Eclipse

In Eclipse, you can configure Lint's behavior via Eclipse's Preferences dialog. Go into Android > Lint Error Checking to see your available options:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

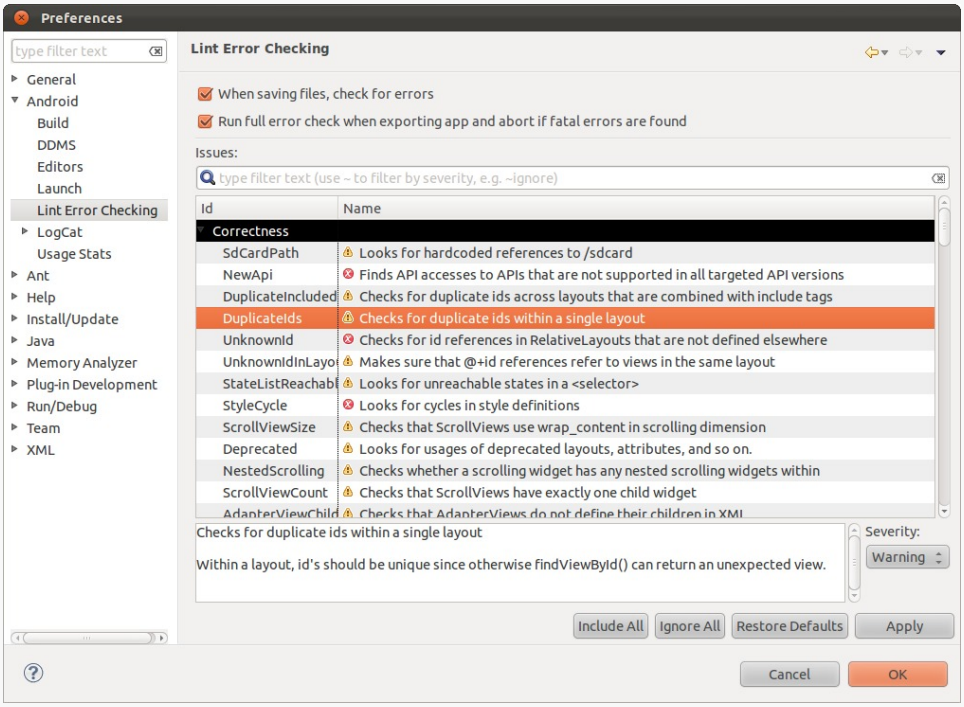


Figure 736: Eclipse Lint Error Checking Preferences

In addition to configuring the automatic Lint checks (e.g., on each file save), you can change some details about the specific checks that Lint makes:

- the severity of the issue, usually set to Warning or Error
- whether the specific issue should be ignored rather than executed

To change Lint behavior on a per-project basis, go into the project properties, click on the "Android Lint Preferences" category, and you will see a similar table of issues, which you can configure for this specific project.

Also, from the "Lint Warnings" view, you can elect to suppress certain warnings, either for the entire workspace, the entire project, or for the specific file in which the warning is being presented.

Command Line

One way to suppress issues from the command line is to add the `--disable` switch, listing the issues (or categories of issues) to skip. You can use the `--list` switch to see what checks are available:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
$ lint --list
Valid issue categories:
    Correctness
    Correctness:Messages
    Security
    Performance
    Usability:Typography
    Usability:Icons
    Usability
    Accessibility
    Internationalization

Valid issue id's:
"ContentDescription": Ensures that image widgets provide a contentDescription
"FloatMath": Suggests replacing java.lang.Math calls with
      android.util.FloatMath to avoid conversions
"FieldGetter": Suggests replacing uses of getters with direct field access
      within a class
"SdCardPath": Looks for hardcoded references to /sdcard
"NewApi": Finds API accesses to APIs that are not supported in all targeted
      API versions
"DuplicateIncludedIds": Checks for duplicate ids across layouts that are
      combined with include tags
"DuplicateIds": Checks for duplicate ids within a single layout
"UnknownId": Checks for id references in RelativeLayouts that are not defined
      elsewhere
...
```

(where the `...` is simply a truncation of the list shown here, which is very long)

If, for example, you wanted to run `lint` and skip all performance issues, you could use `lint --disable Performance`. If you are uncertain what a particular issue means, the `--show` switch can dump details about the issue:

```
$ lint --show FieldGetter
FieldGetter
-----------
Summary: Suggests replacing uses of getters with direct field access within a
class

Priority: 4 / 10
Severity: Warning
Category: Performance
NOTE: This issue is disabled by default!
You can enable it by adding --enable FieldGetter

Accessing a field within the class that defines a getter for that field is at
least 3 times faster than calling the getter. For simple getters that do
nothing other than return the field, you might want to just reference the
local field directly instead.
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
More information: http://developer.android.com/guide/practices/design/
performance.html#internal_get_set
```

Another option is to create a `lint.xml` file, in the root directory of your project, containing information about which particular issues should be suppressed for that project. The benefit here is that you can configure suppression at a finer granularity, blocking issues for certain files or directories and allowing them for others. The sample `lint.xml` from the Lint documentation looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<lint>
    <!-- Disable the given check in this project -->
    <issue id="IconMissingDensityFolder" severity="ignore" />

    <!-- Ignore the ObsoleteLayoutParam issue in the given files -->
    <issue id="ObsoleteLayoutParam">
        <ignore path="res/layout/activation.xml" />
        <ignore path="res/layout-xlarge/activation.xml" />
    </issue>

    <!-- Ignore the UselessLeaf issue in the given file -->
    <issue id="UselessLeaf">
        <ignore path="res/layout/main.xml" />
    </issue>

    <!-- Change the severity of hardcoded strings to "error" -->
    <issue id="HardcodedText" severity="error" />
</lint>
```

You can also have a similar `lint.xml` file that you use outside of any project, by passing in the `--config` switch pointing to it.

Note that it is unclear how these switches would be provided to the stock Gradle for Android `lint` tasks.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Using Hierarchy View

Android comes with a Hierarchy View tool, designed to help you visualize your layouts as they are seen in a running activity in a running emulator. So, for example, you can determine how much space a certain widget is taking up, or try to find where a widget is hiding that does not appear on the screen.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Launching Hierarchy View

To use the Hierarchy View, you first need to fire up your emulator, install your application, launch your activity, and navigate to the spot you wish to examine. Note that you cannot use Hierarchy View with a production Android device without some help.

To launch Hierarchy View, you have two options:

1. From Eclipse, open the Hierarchy View perspective
2. Use the Android Device Monitor. In Android Studio, this is available from the Tools > Android menu. From the command line, run the `monitor` program to bring up the Android Device Monitor. In the Android Device Monitor, choose Window > Open Perspective from the main menu, and open Hierarchy View:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

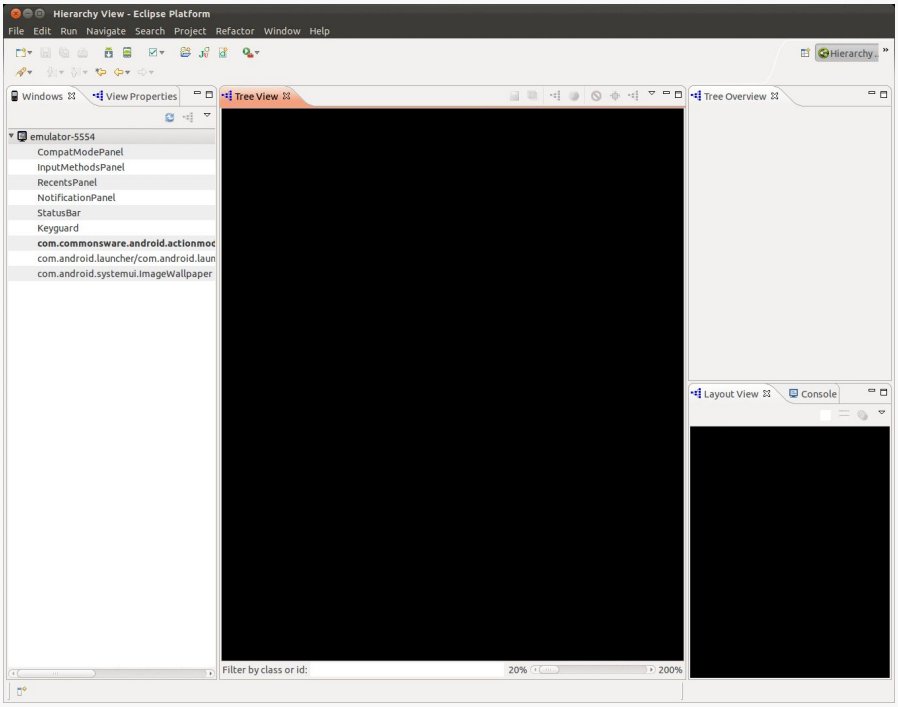


Figure 737: Hierarchy View, in Eclipse, As Originally Opened

The roots of the tree-table on the left show the emulator instances presently running on your development machine. The leaves represent applications running on that particular emulator. Your activity will be identified by application package and class (e.g., `com.commonsware.android.files/...`).

Viewing the View Hierarchy

Where things get interesting, though, is when you double-click on your activity in the tree-table. After a few seconds, the details spring into, er, view:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

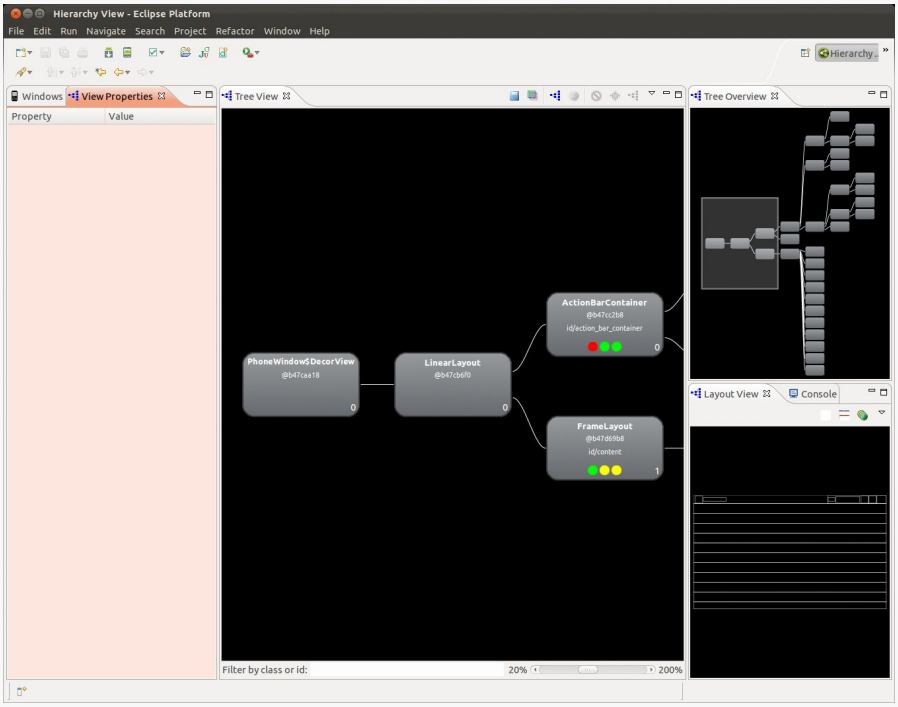


Figure 738: Hierarchy View, in Eclipse, Showing an Activity

The main area of the Layout View shows a tree of the various widgets and stuff that make up your activity, starting from the overall system window and driving down into the individual UI widgets that users are supposed to interact with. This includes both widgets and containers defined by your application and others that are supplied by the system, including the title bar.

Clicking on one of the views adds more information to this perspective:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

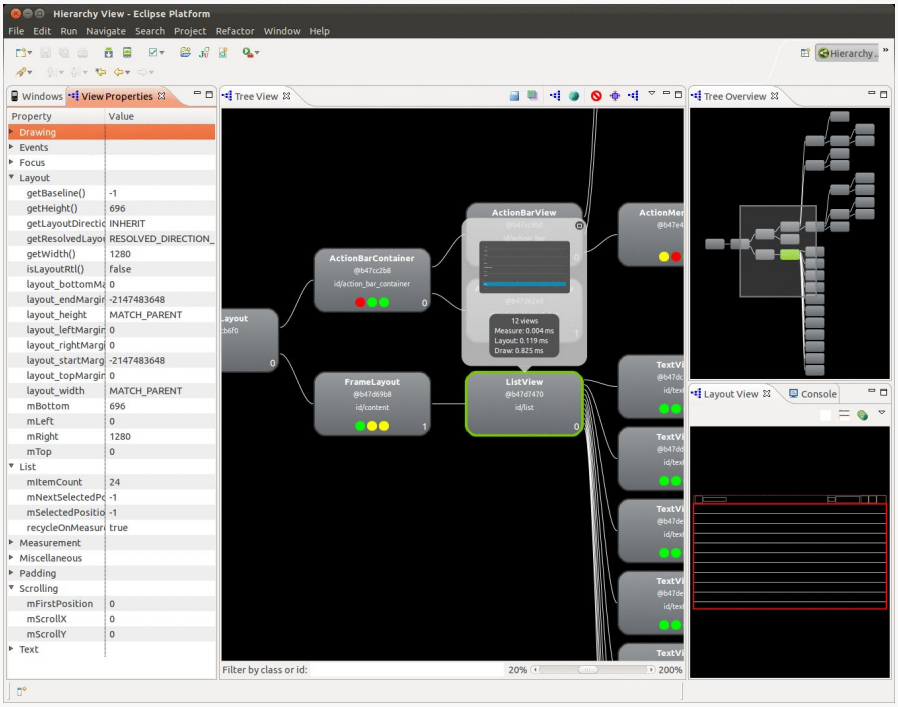


Figure 739: Hierarchy View, in Eclipse, Showing a View's Details

Now, we get:

- In the left region of the Viewer, we see the properties of the selected widget or container, in its own tree-table.
- In the Tree View in the middle, the selected widget or container has a pop-up bubble with what that particular `View` looks like on the screen, along with some performance timing data.
- In the Tree Overview in the upper-right portion of the tool, our selected `View` is highlighted in green.
- In the Layout View in the lower-right portion of the tool, our selected `View` is highlighted in red in the wireframe.

From the toolbar above the Tree View, you can:

- Save the tree diagram as a PNG file
- Save the UI as a Photoshop PSD file, with different layers for the different widgets and containers
- Force the UI to repaint in the emulator or re-load the hierarchy, in case you have made changes to a database or to the app's contents and need a fresh diagram

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

ViewServer

One major limitation of Hierarchy View is that it only works with the emulator by default. There is no means for it to pull information from random activities running on production hardware.

However, Romain Guy, one of the core Android engineers, has published a `ViewServer` open-source component that gets around this limitation.

If you add the `ViewServer` source code to your project, and register your activities as they are created (and remove them when they are destroyed), you will be able to use Hierarchy View with them. However, this is a bit dangerous on a production app, so you should strongly consider using `BuildConfig.DEBUG` to only enable this logic in debug builds.

Blending in the `BuildConfig.DEBUG` concept with Mr. Guy's supplied sample usage, we get something like this:

```
public class MyActivity extends Activity {
   public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);

     // Set content view, etc.

     if (BuildConfig.DEBUG) ViewServer.get(this).addWindow(this);
   }

   public void onDestroy() {
     if (BuildConfig.DEBUG) ViewServer.get(this).removeWindow(this);

     super.onDestroy();
   }

   public void onResume() {
     super.onResume();

     if (BuildConfig.DEBUG) ViewServer.get(this).setFocusedWindow(this);
   }
}
```

Also note that `ViewServer` requires that your application hold the `INTERNET` permission, which you may already have requested for other reasons.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Using DDMS

Another tool in the Android developer's arsenal is the Dalvik Debug Monitor Service (DDMS). This is a "Swiss army knife", allowing you to do everything from browse log files, update the GPS location provided by emulator, simulate incoming calls and messages, and browse the on-emulator storage to push and pull files.

We have already seen the use of DDMS for viewing your logs via the LogCat view. This chapter will explore a few other uses of DDMS beyond LogCat.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the chapter on using LogCat.

While not strictly a prerequisite, you will find detailed coverage of other features of DDMS in the chapter on memory leak analysis using MAT and the chapter on measuring bandwidth consumption.

Starting DDMS

As a reminder, to launch DDMS, you have two options:

1. From Eclipse, choose the DDMS perspective
2. From the command line, run the `monitor` program to bring up the Android Device Monitor — the DDMS perspective should appear by default

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

DDMS will initially display a tree of emulators and devices and the running programs on each. Clicking on an emulator or device allows you to use the rest of the tools to work with that specific Android environment.

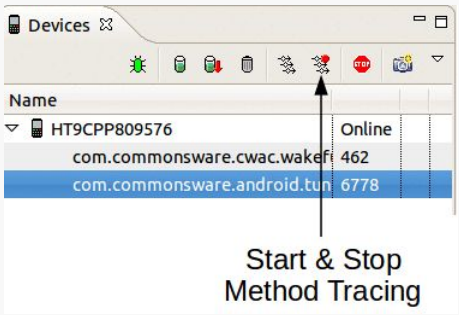


Figure 740: DDMS, with Emulator Selected

File Push and Pull

The File Explorer view in DDMS allows you to upload and download files from your selected device or emulator. The view shows a typical file explorer-type tree of available folders and files on your selected device or emulator, which you can navigate as you would similar sorts of explorers you have no doubt seen elsewhere.

The toolbar above the view gives you three choices, once you have a folder or file selected:

- Push a file to the device, either into a selected folder or to replace a selected file
- Pull a file from the device to your development machine
- Delete a selected file

There are a few caveats to this:

1. You cannot pull or delete a folder.
2. You cannot create directories through this tool. You will either need to use `adb shell` or create them from within your application.
3. While you can putter through most of the files on an emulator, you can access very little outside of `/mnt/sdcard` on an actual device, due to Android security restrictions.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Screenshots

To take a screenshot of the Android emulator or device, click on the camera icon in the toolbar in the Devices view. This will bring up a dialog box containing an image of the current screen:

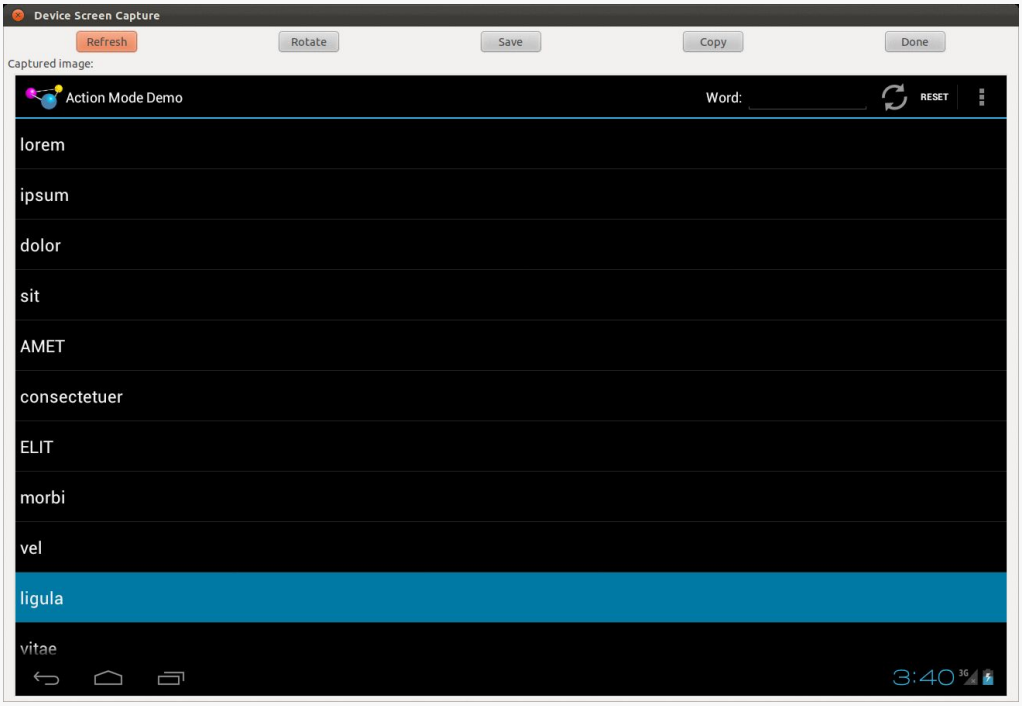


Figure 741: DDMS Screen Capture Dialog

From here, you can click “Save” to save the image as a PNG file somewhere on your development machine, “Refresh” to update the image based on the current state of the emulator or device, “Rotate” to change the orientation of the screenshot, or “Done” to close the dialog.

Location Updates

To use DDMS to supply location updates to your application, the first thing you must do is have your application use the GPS `LocationProvider`, as that is the one that DDMS is set to update.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Then, in the Emulator Control view, you will see a Location Controls section. Here, you will find a smaller tabbed pane with three options for specifying locations: Manual, GPX, and KML:

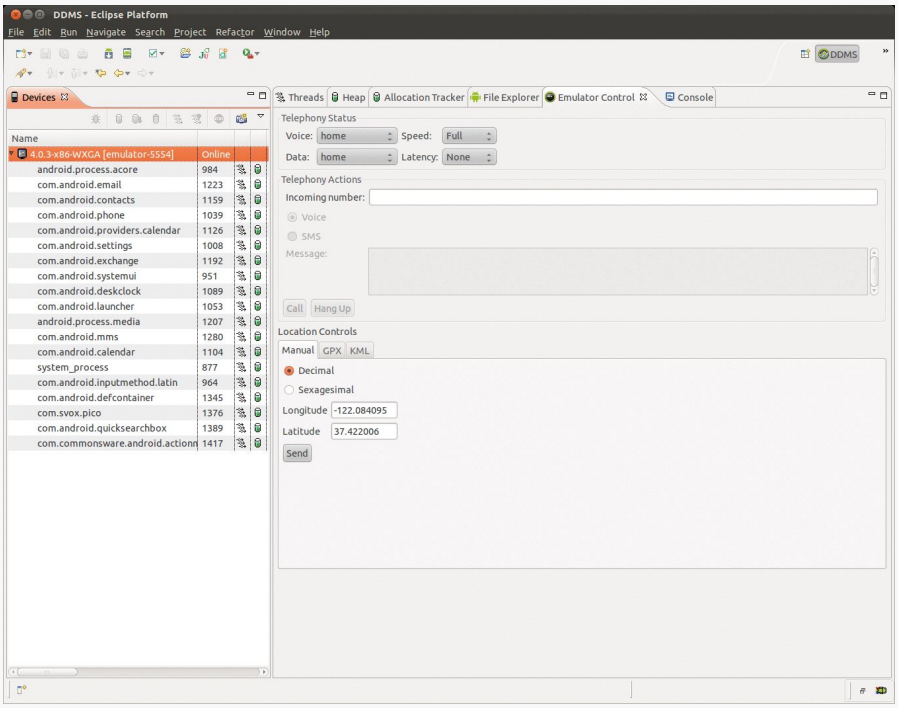


Figure 742: DDMS Emulator Control View

The Manual tab is fairly self-explanatory: provide a longitude and latitude, in decimal degrees, and click the Send button to submit that location to the emulator. The emulator, in turn will notify any location listeners of the new position. The fields are pre-populated with the longitude and latitude of a building on Ampitheater Parkway, in Mountain View, CA, USA.

Note that you cannot simulate GPS on a device this way, only on an emulator.

Placing Calls and Messages

If you want to simulate incoming calls or SMS messages to the Android emulator, DDMS can handle that as well.

On the Emulator Control view, above the Location Controls group, is the Telephony Actions group.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To simulate an incoming call, fill in a phone number, choose the Voice radio button, and click Call. At that point, the emulator will show the incoming call, allowing you to accept it (via the green phone button) or reject it (via the red phone button):

To simulate in an incoming text message, fill in a phone number, choose the SMS radio button, enter a message in the provided text area, and click Send. The text message will then be delivered to the emulator as if it came in over the air.

Note that you cannot simulate SMSes on a device this way, only on an emulator.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Tuning Android Applications

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Issues with Speed

Mobile devices are never fast enough. Either they are slow in general (e.g., slow CPU) or they are slow for particular operations (e.g., advanced game graphics).

What you do not want is for your application to be unnecessarily slow, where the user determines what is and is not "necessary". Your opinion of what is "necessary", alas, is of secondary importance.

This part of the book will focus on speed, including how you can measure and reduce lag in your applications. First, though, let's take a look at some of the specific issues surrounding speed.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

Getting Things Done

In some cases, you simply cannot seem to get the work done that you want to accomplish. Your database query seems slow. Your encryption algorithm seems slow. Your image processing logic seems slow. And so on.

The limits of the device will certainly make this more of a problem than it might otherwise be. Even a current-era dual-core device will be slow compared to your average notebook or desktop. Also, this sort of speed issue is pervasive throughout computing, with decades of experience to help developers learn how to write leaner code.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This part of the book will aim to help you identify where the problem spots are, so you know what needs optimization, and then some Android-specific techniques for trying to improve matters.

Your UI Seems… Janky

Sometimes, the speed would be less of an issue for the user, if it was not freezing the UI or otherwise making it appear sluggish and "janky".

The Android widget framework operates in a single-threaded mode. All UI changes — from setting the text of a `TextView` to handling scrolling of a `GridView` — are processed as events on an event queue by the main application thread. That same thread is used for most UI callbacks, including activity lifecycle methods (e.g., `onCreate()`) and UI event methods (e.g., `onClick()` of a `Button`, `getView()` of an `Adapter`). Any time you take in those methods on the main application thread tie up that thread, preventing it from processing other GUI events or dispatching user input. For example, if your `getView()` processing in an `Adapter` takes too long, scrolling a `ListView` may appear slow compared to other `ListView` widgets in other applications.

Your objective is to identify where things are slow and move them into background operations. Some of this has been advised since the early days of Android, such as moving all network I/O to background threads. Some of this has arisen more recently, such as the move to use the "loader" framework to help you get data from data stores in the background for populating your UI.

This part of the book will point out ways for you to find out where you may be doing unfortunate things on the main application thread and techniques for getting that work handled by a background thread, or possibly eliminated outright.

Not Far Enough in the Background

Sometimes, even work you are trying to do in the background will seem to impact the foreground.

For example, you might think that your `Service` is automatically in the background. An `IntentService` does indeed use a background thread for processing commands via `onHandleIntent()`. However, all lifecycle methods of any `Service`, including `onStartCommand()`, are called on the main application thread. Hence, any time you take in those lifecycle methods will steal time away from GUI processing for the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

main application thread. The same holds true for `onReceive()` of a `BroadcastReceiver` and all the main methods of a `ContentProvider` (e.g., `query()`).

Even your background threads may not be sufficiently in the background. A process runs with a certain priority, using Linux process management APIs, based upon its state (e.g., if there is an activity in the foreground, it runs at a higher priority than if the process solely hosts some service). This will help to cap the CPU utilization of the background work, but only to a point. Similarly, threads that you fork — directly or via something like `IntentService` — may run at default priority rather than a lower priority. Even with lower priorities for the thread or process, every CPU instruction executed in the background is one clock tick that cannot be utilized by the foreground.

This part of the book will help you identify where you are taking lots of time on various threads and will help you manually manage priorities to help minimize the foreground impact of those threads, in addition to helping you reduce the amount of work those threads have to do.

Playing with Speed

Games, more so than most other applications, are highly speed-dependent. Everyone is seeking the "holy grail" of 60 frames per second (FPS) necessary for smooth animated effects. Not achieving that frame rate overall may mean the application will not appear quite as smooth; sporadically falling below that frame rate will result in jerky animation effects, much like the "janky" UIs in a non-game Android application.

For example, a classic problem with Android game development is garbage collection (GC). Only since the Gingerbread release of Android is the garbage collector concurrent, meaning that it runs in tandem with application code on a parallel thread. Historically, the Android garbage collector was a "stop the world" implementation, that would freeze the game long enough for a bit of GC work to be done before the game could continue. This behavior pretty much guaranteed sporadic failures to maintain a consistent frame rate. This caused game developers to have to take particular steps to avoid generating any garbage, such as maintaining its own object pools, to minimize or eliminate garbage collection pauses.

This book does not focus much on specific issues related to game development, though many of the techniques outlined here will be relevant for game developers.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Finding CPU Bottlenecks

CPU issues tend to manifest themselves in three ways:

- The user has a bad experience when using your app directly — scrolling is sluggish, activities take too long to display, etc.
- The user has a bad experience when your app is running in the background, such as having slower frame rates on their favorite game because you are doing something complex in a service
- The user has poor battery performance, driven by your excessive CPU utilization

Regardless of how the issue appears to the user, in the end, it is a matter of you using too much CPU time. That could be simply because your application is written to be constantly active (e.g., you have an everlasting service that uses `TimerTask` to wake up every second and do something). There is little anyone can do to help that short of totally rethinking the app's architecture (e.g., switch to `AlarmManager` and allow the user to configure the polling period).

However, in many cases, the problem is that you are using algorithms – yours or ones built into Android — that simply take too long when used improperly. This chapter will help you identify these bottlenecks, so you know what portions of your code need to be optimized in general or apply the techniques described in later chapters of this part of the book.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate. Reading the introductory chapter to this trail is also a good idea.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Traceview

The #1 tool in your toolbox for finding out where bottlenecks are occurring in your application is Traceview. This is available both within the Eclipse environment — though not as a separate perspective — and as a standalone tool.

What Is Traceview?

Traceview is Android's take on a method profiler. Profilers have existed for most other platforms, in one form or fashion, dating back to the mainframe days.

Technically, the profiling in Android is performed by the Dalvik virtual machine, under the direction of either DDMS or requests from your application code. Dalvik will write the "trace data" (call graphs showing methods, what they call, and the amount of time in each) to a file on external storage of the device or emulator. Traceview then views these trace files in a GUI, allowing you to visualize "hot spots", drill down to find where the time is being taken, and so forth.

At the time of this writing, Traceview is designed for use on single-core devices. Results on multi-core devices may be difficult to interpret.

Collecting Trace Data

Hence, the first step for finding where your CPU bottlenecks lie comes in the form of collecting trace data, to analyze with Traceview. As mentioned, there are two approaches for requesting trace data be logged: using the `Debug` class, and using DDMS.

Debug Class

If you know what chunk of code you want to profile, one way to arrange for the profile is to call `startMethodTracing()` on the `Debug` class. This takes the name of a trace file as a parameter and will begin recording all activity to that file, stored in the root of your external storage. You need to call `stopMethodTracing()` at some point to stop the trace — failing to do so will leave you with a corrupt trace file in the end.

Note that your application will need the `WRITE_EXTERNAL_STORAGE` permission for this to work. If your application does not normally need this permission, make yourself a note to remove it before you ship the production edition of your product, as there is no sense asking for any more permissions than you absolutely need. Also,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

your device or emulator will need enough external storage to hold the file, which can get very large for long traces — 100MB a minute is well within reason.

DDMS

Alternatively, you can initiate tracing via a toolbar button in DDMS. In both the DDMS perspective in Eclipse and the standalone DDMS, there is a button in the toolbar above the tree-table of devices and processes that toggles tracing on and off:

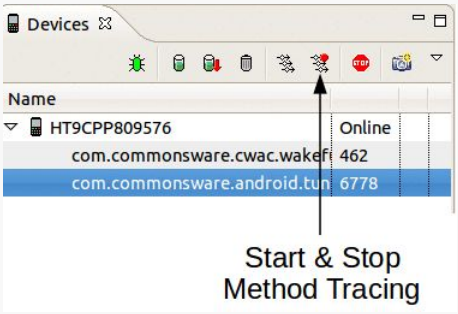


Figure 743: Toolbar button to start and stop method tracing

On Android 2.1 and earlier, this will write the trace out to a file on external storage, much as `startMethodTracing()` does. Hence, your application will need `WRITE_EXTERNAL_STORAGE` in this case, plus have enough external storage space to hold the file.

On Android 2.2 and newer, though, this data is written straight to the development machine, bypassing external storage. This means you do not need to worry about permissions or free space on your external storage. Hence, unless your problem only exists on Android 2.1 and earlier, you may find it easier to do your Traceview work on a newer Android device or emulator image. The file will wind up in your development machine's temporary directory (e.g., `/tmp` on Linux).

Performance While Tracing

Writing out each method invocation to a trace file adds significant overhead to your application. Run times can easily double or more. Hence, absolute times while tracing is enabled are largely meaningless — instead, as you analyze the data in Traceview, the goal is to examine relative times (i.e., such-and-so method takes up X% of the CPU time shown in the trace).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Also, running Traceview disables the JIT engine in Dalvik, further harming performance. Notably, this will not affect any native code you have added via the NDK, so an application run in Traceview will give you unusual results (much worse Java performance, more normal native performance).

Displaying Trace Data

Given that we have collected a trace file with data, the next step is to open up Traceview on that file. Depending on how you collected the file, Traceview may appear "automagically", or it may require you to manually start it up and point it to the trace file.

Eclipse/DDMS

If you used the DDMS perspective in Eclipse to record the trace data, the Debug perspective in Eclipse will automatically open up when you stop the tracing, showing you a Traceview tool:

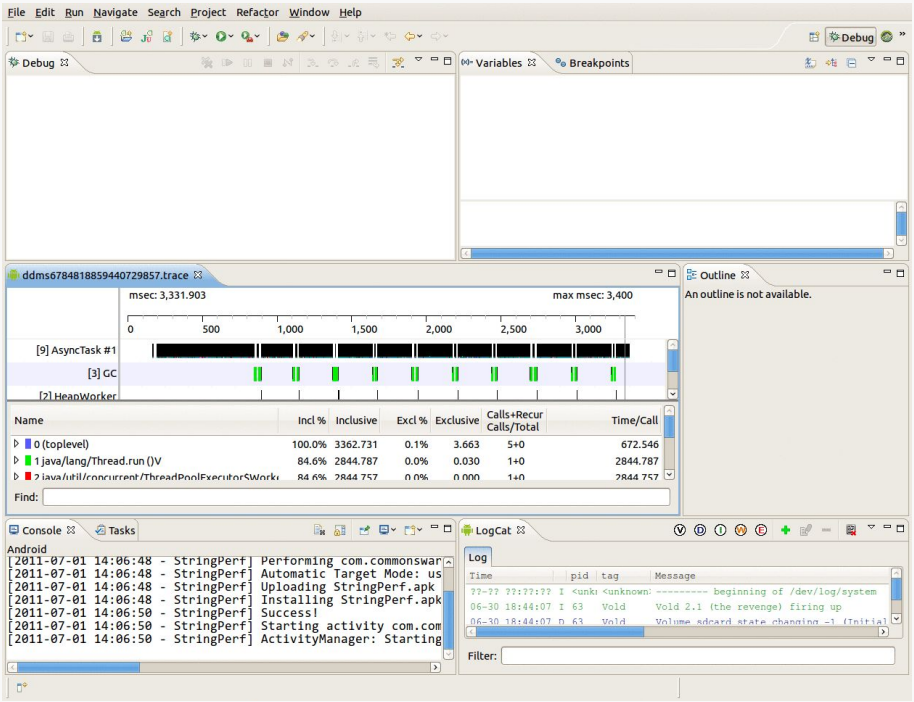


Figure 744: Debug perspective in Eclipse showing Traceview (middle left)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Standalone Traceview

If you used standalone DDMS and run a trace on Android 2.2 and up, it will automatically launch in the standalone Traceview utility.

If your trace file wound up on external storage on your device or emulator, you will need to download it to your development machine, whether using the File Manager within DDMS, or via the `adb pull` command. Once on your development machine, you can view it in the standalone Traceview tool using the `traceview` command:

```
traceview <path-to-trace-file>
```

Or, you can import the file into your Eclipse project, then double-click on it in the Project Explorer to view it in the Traceview tool.

Interpreting Trace Data

Of course, the challenge is in making sense of what Traceview is trying to present.

For example, a classic performance bug in Java development is using string concatenation:

```
package com.commonsware.android.traceview;

import android.view.View;
import android.widget.TextView;

public class StringConcatActivity extends BaseActivity {
    StringConcatTask createTask(TextView msg, View v) {
      return(new StringConcatTask(msg, v));
    }

    class StringConcatTask extends BaseTask {
      StringConcatTask(TextView msg, View v) {
        super(msg, v);
      }

      protected String doTest() {
        String result="This is a string";

        result+=" -- that varies --";
        result+=" and also has ";
        result+=String.valueOf(4);
        result+=" hyphens in it";

        return(result);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
        }
    }
}
```

Here is a Traceview screen showing that code executed 100,000 times, as packaged in a `StringPerfConcat` activity in the `Tuning/Traceview` sample project:

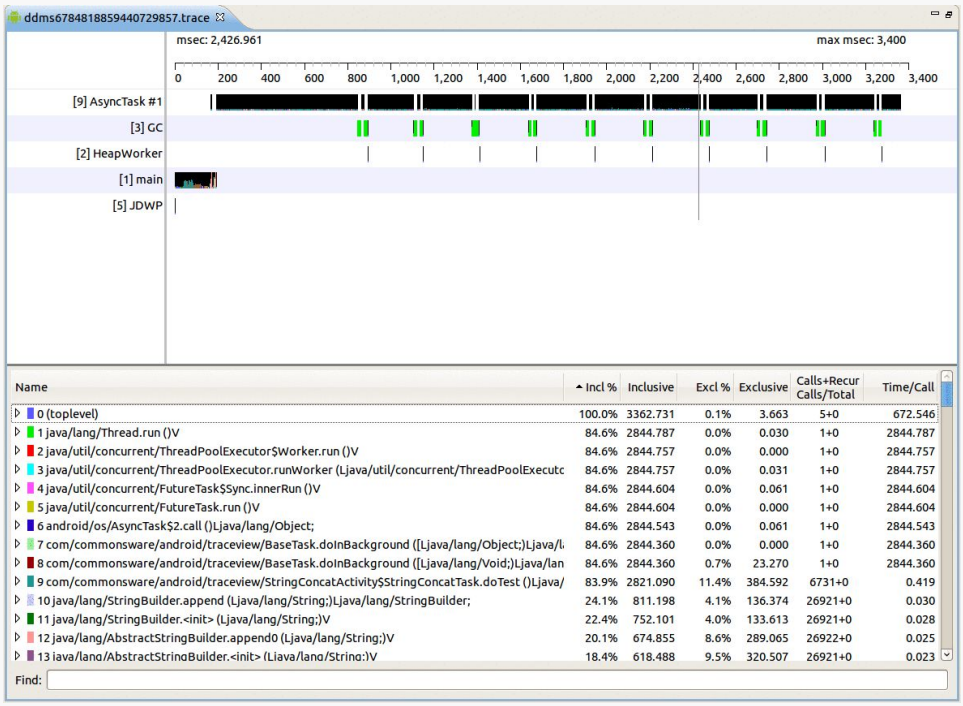


Figure 745: Expanded look at Traceview tool

The bars in the top portion of the display show different threads in the running application, in a timeline fashion, with time running from left to right. The "main" bar shows the main application thread, spending most of its time initializing the activity. The GC and HeapWorker threads are involved in garbage collection, popping in from time to time to collect garbage during 100,000 iterations of the above algorithm. Those 100,000 iterations are run in an `AsyncTask`, so we do not encounter an application-not-responding (ANR) dialog, and that is the "AsyncTask #1" thread at the top of the diagram.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

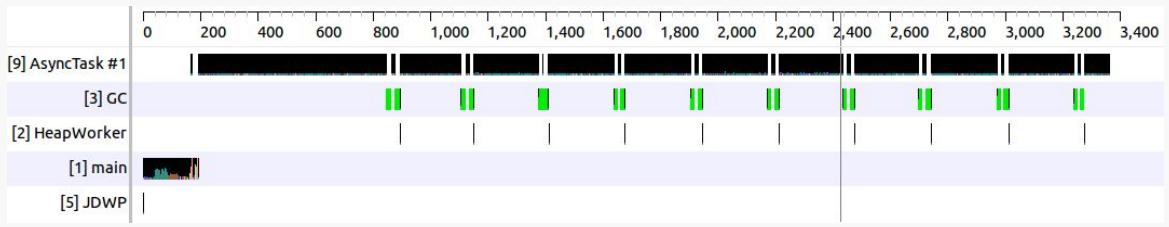


Figure 746: Zoomed in look at the TraceView thread timelines

You will notice that the horizontal timeline bars are not contiguous – there are gaps. In fact, if you were to combine all of the timelines into one, the "holes" in most of the rows would be filled by time in another row. This is illustrating that there is only one core on most Android device CPUs (these images were taken from a test run on a single-core Nexus One). We think of `AsyncTask` as moving work to the background, but it is important to remember that it still is consuming CPU time, even if the background thread means that we are not tying up the main application thread.

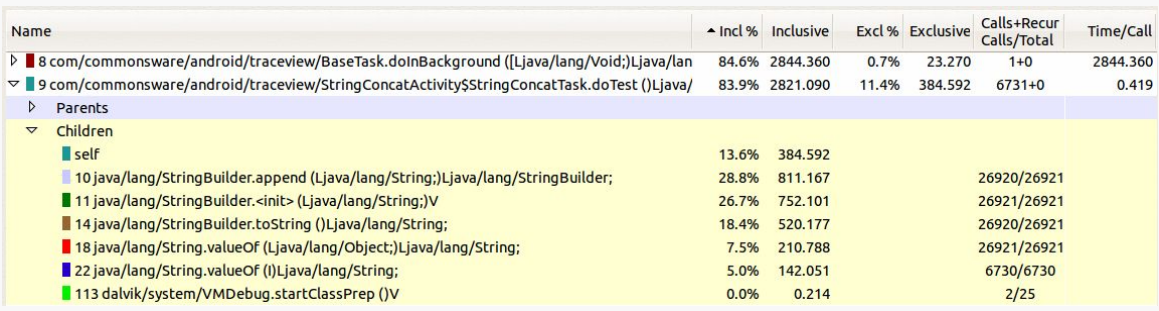
The bottom half of the display shows what methods are taking up all of the time, inclusively, in descending order. By "inclusively", Traceview means "code executed in this method and any methods it invokes". Hence, the top "100.0%" line shows the entry point to the whole application, and the next line shows where the `AsyncTask`'s background thread is being forked, and so on.

Typically, you want to find lines that reference your code. In this case, lines 7–9 are from the `com.commonsware` package. Let's focus on those:

Figure 747: Sample application method calls in Traceview

On their own, these lines are not especially informative. However, if we fold open the bottom row, using the arrow indicator on the left, we can drill down into what is going on inside that particular method, which happens to be the algorithm shown earlier in this section:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition



Name	Incl %	Inclusive	Excl %	Exclusive	Calls+Recur Calls/Total	Time/Call
8 com/commonsware/android/traceview/BaseTask.doInBackground ([Ljava/lang/Void;)Ljava/lan	84.6%	2844.360	0.7%	23.270	1+0	2844.360
9 com/commonsware/android/traceview/StringConcatActivity$StringConcatTask.doTest ()Ljava/	83.9%	2821.090	11.4%	384.592	6731+0	0.419
Parents						
Children						
self	13.6%	384.592				
10 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder;	28.8%	811.167			26920/26921	
11 java/lang/StringBuilder.<init> (Ljava/lang/String;)V	26.7%	752.101			26921/26921	
14 java/lang/StringBuilder.toString ()Ljava/lang/String;	18.4%	520.177			26920/26921	
18 java/lang/String.valueOf (Ljava/lang/Object;)Ljava/lang/String;	7.5%	210.788			26921/26921	
22 java/lang/String.valueOf (I)Ljava/lang/String;	5.0%	142.051			6730/6730	
113 dalvik/system/VMDebug.startClassPrep ()V	0.0%	0.214			2/25	

Figure 748: Drilling down in Traceview

The "self" line refers to code that is directly executed in the method, not involving a nested method call, such as variable declarations and returning values. We see the `valueOf()` calls, along with three rows showing references to `StringBuilder`. On the surface, that may seem odd, considering that we are not referring to `StringBuilder` in the source code.

It turns out that the `javac` compiler replaces string concatenation with `append()` calls on a `StringBuilder`, created on the fly for that specific concatenation. So, of the 83.9% of the time taken up in the entire run by the `doTest()` method, 26.7% is taken up by creating these temporary `StringBuilder` objects, 28.8% is consumed by calling `append()` on the `StringBuilder`, and another 18.4% is used by calling `toString()` to get the resulting `String` out of the `StringBuilder`.

This suggests an optimization: we could create our own `StringBuilder` and use it for concatenating the text, thereby saving us creating a few temporary ones and calling `toString()` extra times:

```
package com.commonsware.android.traceview;

import android.view.View;
import android.widget.TextView;

public class StringBuilderActivity extends BaseActivity {
  StringBuilderTask createTask(TextView msg, View v) {
    return(new StringBuilderTask(msg, v));
  }

  class StringBuilderTask extends BaseTask {
    StringBuilderTask(TextView msg, View v) {
      super(msg, v);
    }

    protected String doTest() {
      StringBuilder result=new StringBuilder("This is a string");
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      result.append(" -- that varies --");
      result.append(" and also has ");
      result.append(String.valueOf(4));
      result.append(" hyphens in it");

      return(result.toString());
    }
  }
}
```

This implementation of the algorithm runs about twice as fast as the first.

The “Exclusive” and “Excl %” columns show how much time is taken in an individual method itself, not including any children. If you sort on that, you see the specific local spots where time is being taken up. For example, here is a Traceview roster from testing the second algorithm shown above (the `StringPerfBuilder` activity):

Name	Incl %	Inclusive	▲ Excl %	Exclusive	Calls+Recur Calls/Total	Time/Call
11 java/lang/AbstractStringBuilder.append0 (Ljava/lang/String;)V	42.2%	1437.766	16.6%	564.832	48805+0	0.029
12 java/lang/String._getChars (II[CI)V	15.7%	533.731	10.6%	360.757	61006+0	0.009
13 dalvik/system/VMDebug.startGC ()V	10.2%	347.656	10.2%	347.656	11+0	31.605
9 com/commonsware/android/traceview/StringBuilderActivity$StringBuilderTask.doTest ()Ljava/	83.1%	2830.446	8.6%	291.807	12201+0	0.232
15 java/lang/AbstractStringBuilder.enlargeBuffer (I)V	10.1%	342.959	7.7%	263.821	24401+0	0.014
18 java/lang/System.arraycopy (Ljava/lang/Object;ILjava/lang/Object;II)V	7.4%	252.688	7.4%	252.688	85509+0	0.003
10 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder;	49.5%	1685.716	7.3%	247.981	48804+0	0.035
16 java/lang/AbstractStringBuilder.<init> (Ljava/lang/String;)V	8.4%	285.216	4.3%	147.095	12201+0	0.023
24 java/lang/String.length ()I	3.9%	134.395	3.9%	134.395	61018+0	0.002
21 java/lang/AbstractStringBuilder.toString ()Ljava/lang/String;	5.1%	174.300	3.8%	129.444	12201+0	0.014
32 java/lang/IntegralToString.convertInt (Ljava/lang/AbstractStringBuilder;I)Ljava/lang/String;	2.0%	69.758	2.0%	69.727	12202+0	0.006
23 java/lang/IntegralToString.intToString (I)Ljava/lang/String;	4.0%	137.850	2.0%	68.123	12201+0	0.011
17 java/lang/String.valueOf (I)Ljava/lang/String;	7.9%	267.410	2.0%	66.429	12201+0	0.022
19 java/lang/StringBuilder.toString ()Ljava/lang/String;	7.1%	240.708	2.0%	66.408	12201+0	0.020

Figure 749: Traceview, sorted by exclusive time

We see that the top three culprits are all Android/Dalvik methods, which we cannot optimize. Instead, the fact that they are taking up so much time is indicative of the fact that we are calling them a lot, also in evidence by the Calls/Total column. You can examine the parents of a call to see where those calls come from, to see if you can change upstream code to result in fewer such calls:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

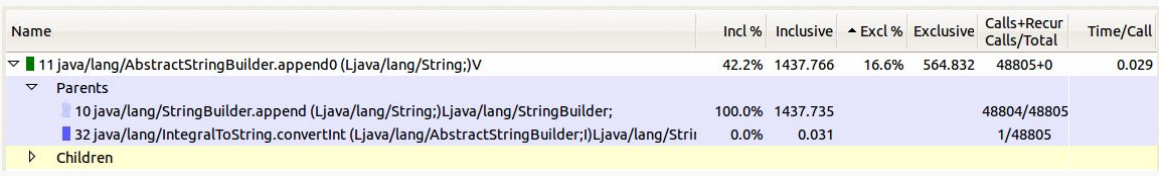


Figure 750: Traceview, showing parents of a method call

Here, we can see that all those `append0()` calls are triggered by calls to `append()` on the `StringBuilder`, which is not terribly surprising.

You can also zoom in to take a very narrow look at the data. Simply click-drag a bar in the timeline to select a region to zoom into. The timeline will switch to show just that range of milliseconds and the calls that take place there:

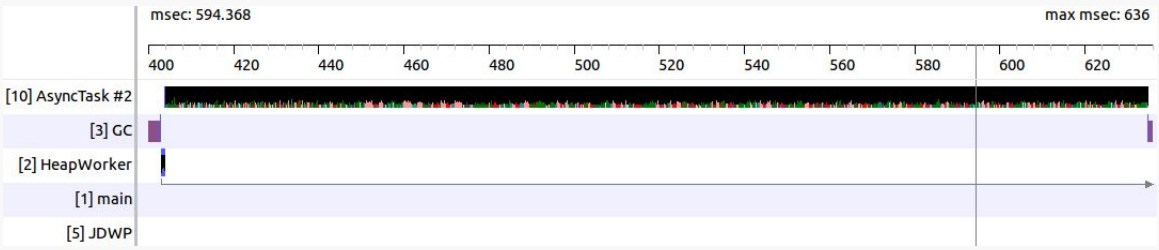


Figure 751: Traceview, zoomed in on ~230 milliseconds of run time

If you zoom in far enough, you will start seeing solid blocks of color, corresponding to the color-coded methods in the table of results on the bottom half of the screen. You can tap on any block of color to bring up that specific method in the table:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

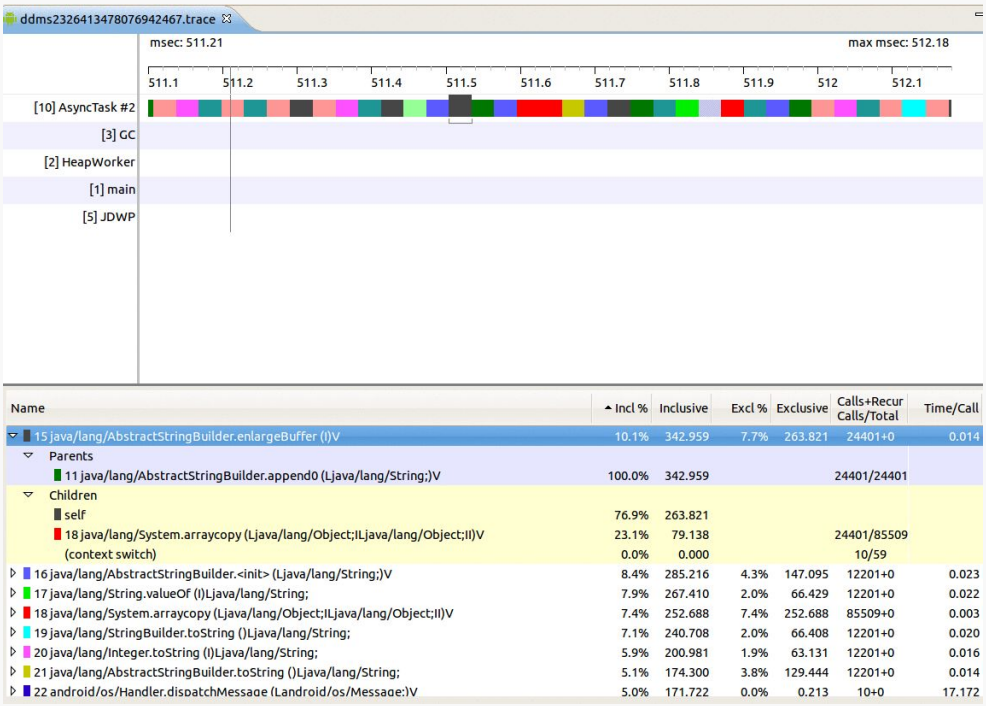


Figure 752: Traceview, zoomed in on ~1 millisecond of run time, highlighting one specific method

Zooming back out, though, is somewhat of a pain. If you drag the timeline itself (not one of the bars, but the "meter stick" showing the milliseconds) from left to right, you will zoom out. Do this enough times, and you can return approximately to the original state.

Other General CPU Measurement Techniques

While Traceview is great for narrowing down a general performance issue to a specific portion of code, it does assume that you know approximately where the problem is, or that you even have a problem in the first place. There are other approaches to help you identify if and (roughly) where you have problems, which you can then attack with Traceview to try to refine.

Logging

Traceview can be useful, if you have a rough idea of where your performance problem lies and need to narrow it down further. If you have a large and complicated application, though, trying to sift through all of it in Traceview may be difficult.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, there is nothing stopping you from using good old-fashioned logging to get a rough idea of where your problems lie, for further analysis via Traceview. Just sprinkle your code with `Log.d()` calls, logging `SystemClock.uptimeMillis()` with an appropriate label to identify where you were at that moment in time. "Eyeballing" the LogCat output can illustrate areas where unexpected delays are occurring — the areas in which you can focus more time using Traceview.

A useful utility class for this is `TimingLogger`, in the `android.util` package. It will collect a series of "splits" and can dump them to LogCat along with the overall time between the creation of the `TimingLogger` object and the corresponding `dumpToLog()` method call. Note, though, that this will only log to LogCat when you call `dumpToLog()` — all of the calls to `split()` to record intermediate times have their results buffered until `dumpToLog()` is called. Also note that logging needs to be set to VERBOSE for this information to actually be logged — use the command `adb shell setprop log.tag.LOG_TAG VERBOSE`, substituting your log tag (supplied to the `TimingLogger` constructor) for `LOG_TAG`.

FPS Calculations

Sometimes, it may not even be strictly obvious how bad the problem is. For example, consider scrolling a `ListView`. Some performance issues, like sporadic "hiccups" in the scrolling, will be visually apparent. However, absent those, it may be difficult to determine whether your particular `ListView` is behaving more slowly than you would expect.

A classic measurement for games is frames per second (FPS). Game developers aim for a high FPS value — 60 FPS is considered to be fairly smooth, for example. However, this sort of calculation can only really be done for applications that are continuously drawing – such as Romain Guy's `WindowBackground` sample application. Ordinary Android widget-based UIs are only drawing based upon user interaction or, possibly, upon background updates to data. In other words, if the UI will not even be trying to draw 60 times in a second, trying to measure FPS to get 60 FPS is pointless.

You may be able to achieve similar results, though, simply by logging how long it takes to, say, fling a list (use `setOnScrollListener()` and watch for `SCROLL_STATE_FLING` and other events).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

UI "Jank" Measurement

A user interface is considered "janky" if it stutters or otherwise fails to operate smoothly, particularly during animated effects like scrolling. Sometimes, janky behavior is obvious to all. Sometimes, janky behavior is only noticeable to those sensitive to small hiccups in the UI.

This section will outline what "jank" is and how to determine, concretely, if your UI suffers from it.

What, Exactly, is Jank?

Prior to Android 4.0, it was difficult to come up with a concrete definition of jank. In effect, we were stuck with "I know it when I see it" ad-hoc analysis, rather than being able to rely on concrete measurements.

Project Butter changed that.

Android 4.0 ties all graphic operations to a 60 frames-per-second "vsync" frequency. If everything is working smoothly, your UI will update 60 times per second, uniformly (versus varying amounts of times between changes).

The converse is also true: if everything is not working smoothly, your UI will not update 60 times per second. This is the source of the term "dropped frames": when the time came around for an update, you were not ready, and that frame was skipped.

There are two main ways in which you will drop a frame:

1. You spend too much time on the main application thread, preventing Android from processing your requested UI updates in a timely fashion
2. Your UI changes are too complex to be rendered before time runs out for the current frame, causing your changes to spill over into the next frame

Each frame is ~16ms in duration on-screen (1/60th of a second). Hence, if we cause per-frame work to exceed 16ms, we will skip, or "drop", a frame.

So, what we need is some way to determine if our code is actually delivering frames on time.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Using gfxinfo

To determine if our problem is in the actual rendering of our UI updates, we can use the GPU profiling feature added in Android 4.2.

Enabling Developer Options

To toggle on GPU profiling, you will need to be able to get to the Developer Options portion of your Settings app. If you see this — typically towards the bottom of the list on the initial Settings screen — just tap on the entry.

If, however, Developer Options is missing, then you will need to use the super-secret trick for enabling Developer Options:

1. Tap on "About Phone", "About Tablet", or the equivalent at the bottom of your Settings list
2. Tap on the "Build Number" entry seven times in succession
3. Press BACK, and "Developer Options" should now be in the list

Toggling on GPU Profiling

There are two checkboxes in Developer Options that need to be checked for GPU profiling to be enabled.

The first is "Force GPU rendering", in the Drawing section. As the name suggests, this will force your application to use the GPU for drawing, even if your application may have requested that hardware acceleration be disabled. Since most applications do not force hardware acceleration to be disabled, this checkbox probably will have no real effect on your app. Note that if you disabled hardware acceleration due to specific rendering problems, this checkbox will probably cause those rendering artifacts to re-appear during your testing.

The second is "Profile GPU rendering", in the Monitoring section. This will cause the device to keep track of graphics performance on a per-process basis, in a way that we can dump later on.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

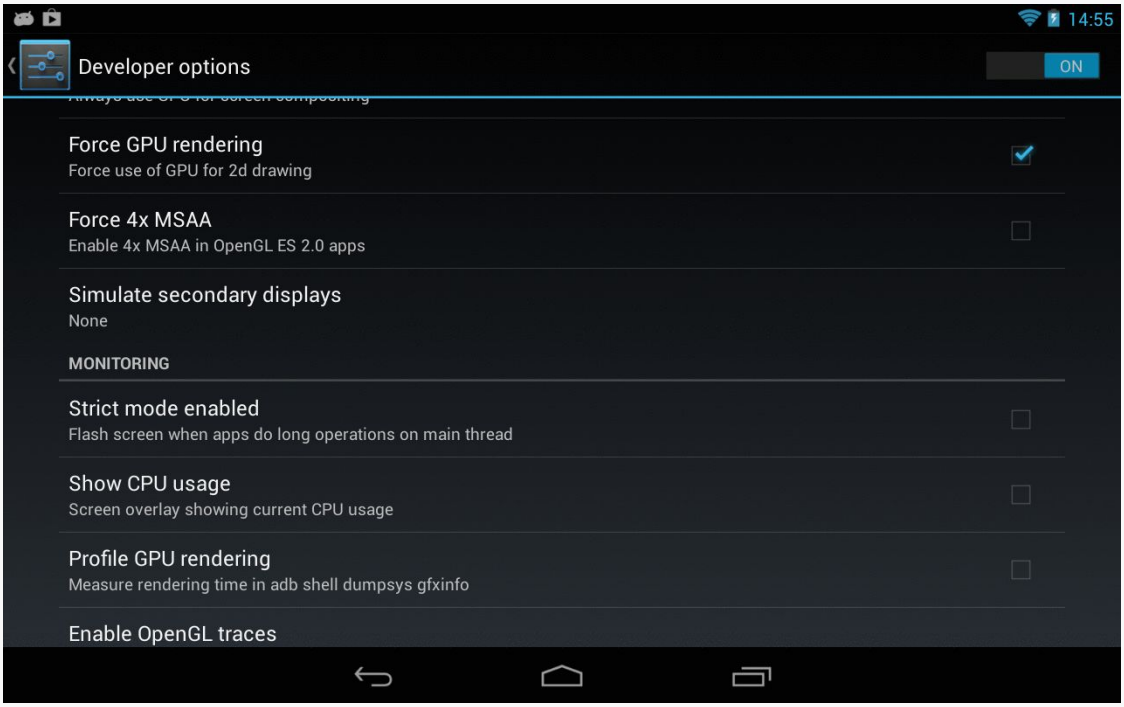


Figure 753: Developer Options, Showing "Force GPU rendering" and "Profile GPU rendering"

If your app was already running, you will need to get rid of its process (e.g., via swiping it off the recent-tasks list) after you check the "Profile GPU rendering" checkbox. At the present time, whether or not this profiling takes effect is determined at process startup time and is not changed on the fly when you toggle the checkbox. Besides, as noted above, starting with a fresh process should give you more accurate results.

Collecting Data

At this point, you can run your app and conduct your specific test, whether manually or via instrumentation (e.g., a targeted JUnit test suite).

When complete, run `adb shell dumpsys gfxinfo ...` in a terminal window, where `...` is replaced by the package name of your app (e.g., `com.commonsware.android.anim.threepane`). This will dump a fair amount of information to the terminal display:

```
mmurphy@xps15:~$ adb shell dumpsys gfxinfo com.commonsware.android.anim.threepane
Applications Graphics Acceleration Info:
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
Uptime: 482460 Realtime: 482454

** Graphics info for pid 3469 [com.commonsware.android.anim.threepane] **

Recent DisplayList operations
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    DrawPatch
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
                  DrawDisplayList
                    Save
                    ClipRect
                    Translate
                    DrawText
                    RestoreToCount
      DrawPatch
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    RestoreToCount

Caches:
Current memory usage / total memory usage (bytes):
  TextureCache          1078032 / 25165824
  LayerCache            7864320 / 16777216
  GradientCache               0 /   524288
  PathCache                   0 /  4194304
  CircleShapeCache            0 /  1048576
  OvalShapeCache              0 /  1048576
  RoundRectShapeCache         0 /  1048576
  RectShapeCache              0 /  1048576
  ArcShapeCache               0 /  1048576
  TextDropShadowCache         0 /  2097152
  FontRenderer 0         262144 /   262144
Other:
  FboCache                    3 /       16
  PatchCache                 89 /      512
Total memory usage:
  9204496 bytes, 8.78 MB

Profile data in ms:

  com.commonsware.android.anim.threepane/
com.commonsware.android.anim.threepane.MainActivity/
android.view.ViewRootImpl@4131e788
  Draw  Process  Execute
  14.45  59.67  10.44
  10.91  1.06  1.20
  1.73  12.80  1.19
  1.45  0.64  0.94
  2.15  0.47  0.57
  0.79  0.50  0.60
  2.23  0.49  0.73
  1.56  0.57  0.52
  6.14  0.47  1.92
  0.84  0.53  0.59
  1.58  0.52  0.60
  1.46  0.55  0.54
  1.74  0.75  0.68
  1.74  0.61  0.61
  1.05  0.62  1.00
  1.05  0.71  1.28
  1.29  0.50  0.56
  2.22  0.60  0.75
  0.90  0.65  1.42
  1.70  0.86  0.61
  0.81  1.07  0.93
  6.66  2.35  0.98
  0.93  5.18  0.73
  0.34  1.24  0.51
  0.45  1.28  0.46
  1.85  4.38  1.45
  1.32  3.15  1.03
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
 1.50  3.16  0.98
 1.42  3.00  1.00
 0.90  2.94  1.00
 0.69  2.36  1.15
 1.08  2.72  0.86
 1.49  4.22  1.49
 0.97  2.91  0.91
 0.89  3.05  0.90
 1.36  3.02  1.07
 1.12  2.95  0.95
 1.63  3.47  1.02
 0.96  2.95  0.98
 2.75  5.55  1.83
 2.11  1.47  0.51
 0.44  1.50  0.48
 0.67  1.46  0.51
 2.07  3.93  3.13
 0.71  4.36  1.93
 1.75  3.31  1.15
 2.39  1.79  1.02
 0.96  1.71  0.81
 0.57  1.70  0.73
 1.88  1.81  0.58
 0.59  1.72  0.55
 2.28  3.74  1.72
 2.66  0.84  0.70
 0.64  0.82  0.64
 0.30  0.80  0.62
 1.78  0.70  0.63
 7.20  2.35  1.04
 0.49  0.21  0.50
 9.99  0.26  0.54
 4.28  0.23  0.66
 0.04  0.26  1.94
 3.55  0.52  0.66
 4.56  0.59  0.62
 5.38  0.33  0.68
 4.44  0.33  0.65
 4.35  0.30  0.73
 3.76  0.27  0.60
 3.72  0.30  0.64
 3.75  0.26  0.58
 4.79  0.33  0.75
 4.68  0.33  0.85
 3.00  0.22  0.53
 2.44  0.26  0.83
14.87  0.69  1.59
 8.68  0.96  1.96
 3.44  0.47  0.96
 3.73  0.22  0.65
 3.06  0.72  0.65
 3.86  0.35  1.13
 3.32  0.26  0.57
 3.21  0.26  0.62
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  3.84  0.26  0.60
  4.85  0.33  0.72
  4.16  0.32  0.70
  3.96  0.30  0.69
  2.60  0.82  0.66
  8.72  0.47  0.69
  0.49  0.31  1.50
  0.46  0.28  0.77
  7.54  3.66  0.90
  7.50  0.27  0.71
  0.06  0.32  2.37
  6.07  0.28  0.97
  3.68  0.27  0.52
  6.39  5.86  4.48
  4.66  0.29  1.28
  0.05  0.26  11.86
  8.87  12.64  1.25
  3.32  0.26  0.58
  6.22  4.77  1.26
  3.49  0.31  0.86
  11.32  10.49  1.26
  10.27  15.09  1.78
  12.50  1.34  2.53
  7.66  4.74  0.58
  0.03  0.24  0.32
  4.43  0.30  0.56
  9.75  2.94  1.68
  17.93  0.47  0.56
  3.81  0.35  1.04
  0.20  2.84  2.72
  10.06  0.28  0.92
  5.74  0.72  1.92
  0.07  0.87  0.53
  2.05  0.95  2.03

View hierarchy:

  com.commonsware.android.anim.threepane/
com.commonsware.android.anim.threepane.MainActivity/
android.view.ViewRootImpl@4131e788
  50 views, 4.48 kB of display lists, 115 frames rendered


Total ViewRootImpl: 1
Total Views:        50
Total DisplayList:  4.48 kB
```

We will discuss what this means in just a bit.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Disabling GPU Profiling

When you are done with your test, it is a good idea to undo the settings changes you made, at least "Profile GPU rendering". That way, the act of collecting this data does not itself add overhead to unrelated tests in the future.

Analyzing the Results

The key bit for our performance analysis is that long table labeled "Profile data in ms:". This reports, for a series of UI requests, how much time is spent:

- drawing your UI changes (e.g., `onDraw()` calls to various widgets and containers)
- processing the low-level drawing commands created via the draw phase, to create the contents of the frame
- executing the frame, sending it to the compositor to display on the screen

One way to interpret this table is to paste it into your favorite spreadsheet program, then use that program to draw a stacked column chart of the data. You can download a spreadsheet in ODS format (for use with LibreOffice, OpenOffice, or other tools that can handle that format) that contains the above table along with a stacked column chart:

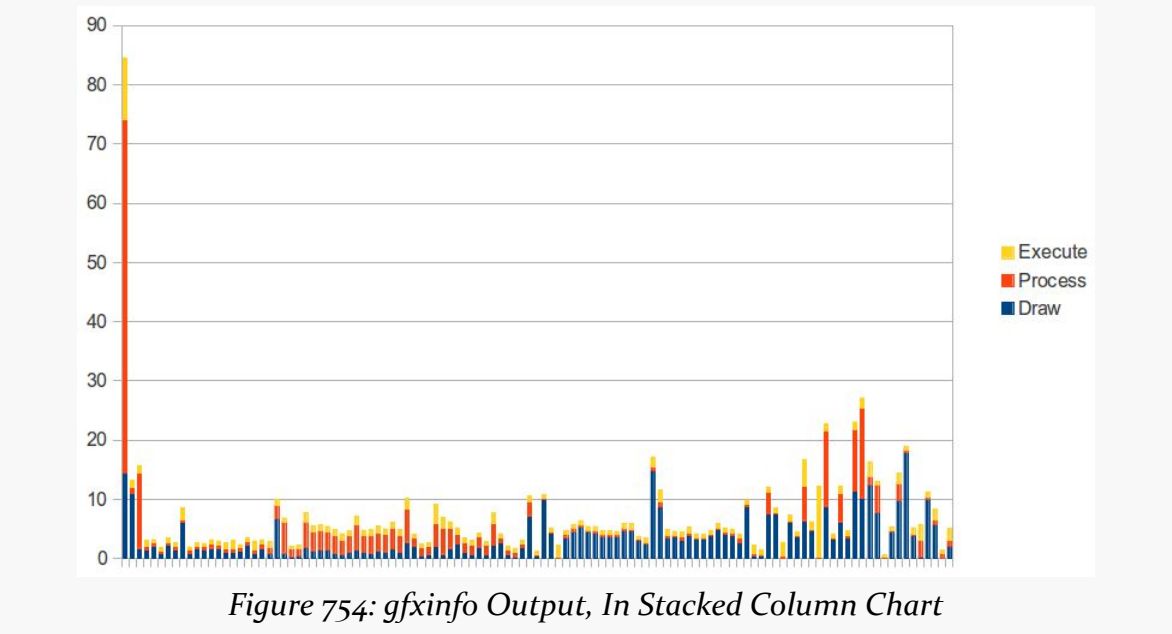


Figure 754: gfxinfo Output, In Stacked Column Chart

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What you are looking for are columns that come close to, or exceed, the 16ms mark, with milliseconds on the Y axis. As you can see, many operations towards the end of the table are near or above 16ms, indicating that we are probably dropping some frames.

Using systrace

Another way we could determine whether or not we are dropping frames is to use `systrace` to collect system-level tracing information about the entire device, including our app.

`systrace` is a very powerful tool, one that 20 or 30 people on the planet truly understand, due to cryptic output and limited documentation. Using `gfxinfo` for detecting dropped frames is simple by comparison. On the other hand, `systrace` works for Android 4.1 and higher, versus the Android 4.2 requirement of `gfxinfo`.

Using `systrace` involves collecting a trace, which is saved in the form of an HTML file. The HTML file is then used to determine what went on during the period of the trace itself.

Enabling and Collecting a Trace: Command-Line

The original means of using `systrace` was from the command line. There is a `systrace.py` Python script located in the `tools/systrace/` directory of your SDK installation. If you have a Python interpreter (e.g., your development machine does not run Windows), you can use this approach.

To indicate what specific bits of information to collect, on Android 4.2 and higher, you can tap the "Enable traces" entry in the Monitoring section of the Developer Options page in Settings. This displays a multi-select dialog of the possible major categories of information that `systrace` should collect:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

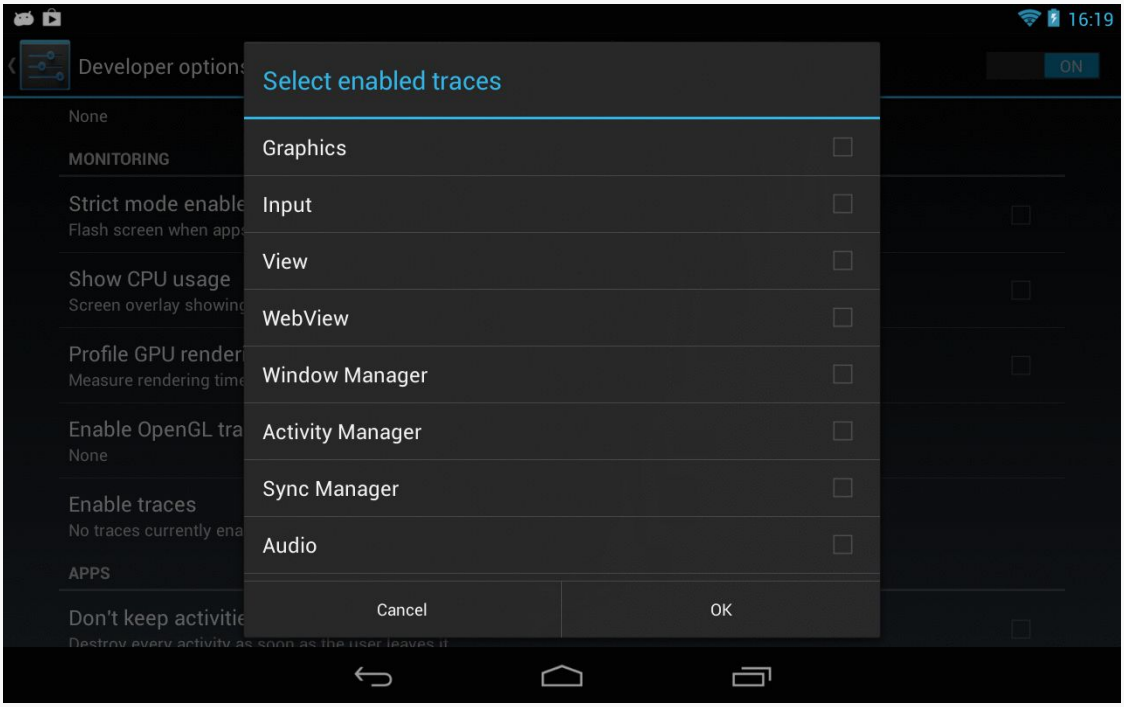


Figure 755: "Enable traces" In Settings

Alternatively, when you run the `systrace.py` script, you can include the `--set-tags` switch, with a comma-delimited list of specific traces ("tags") that you want to collect. The list of available tag names can be found in the developer documentation.

To actually collect the trace, you run the `systrace.py` script, optionally with `--set-tags` or other command-line switches.

On Android 4.1 and 4.2, this would look like:

```
python systrace.py --set-tags gfx,view,wm
adb shell stop
adb shell start
python systrace.py --time=10 -o trace.html
```

The first `python` command runs `systrace.py` just to set the tags to collect. If you set them using Developer Options in Settings, this would not be required. Restarting `adb shell` is apparently needed, for unclear reasons. The second `systrace.py` run will actually collect the trace, for 10 seconds (`--time=10`), resulting in report written to `trace.html` in the current working directory (`-o trace.html`).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The syntax changed for Android 4.3 and higher to simplify matters, combining the two `systrace.py` commands into one:

```
python systrace.py --time=10 -o trace.html gfx view sched wm
```

Note that `--set-tags` is no longer used. Instead, all values not identified by a switch are considered to be tags.

Once you run the script, quickly go to your device and run your test scenario, as the trace starts immediately upon running the script.

Enabling and Collecting a Trace: Eclipse/Monitor

The current version of the ADT plugin for Eclipse also allows you to collect a trace using `systrace`. There is a "Capture system wide trace using systrace" button in the toolbar in the Devices view, typically found in the DDMS perspective:

Figure 756: Systrace Toolbar Button in Devices View

This is also available in the same view from the standalone `monitor` utility, and from Android Studio.

Tapping that toolbar button brings up a dialog that allows you to configure the trace you wish to collect, with checkboxes and fields replacing the variety of command-line switches you might use manually with `systrace.py`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

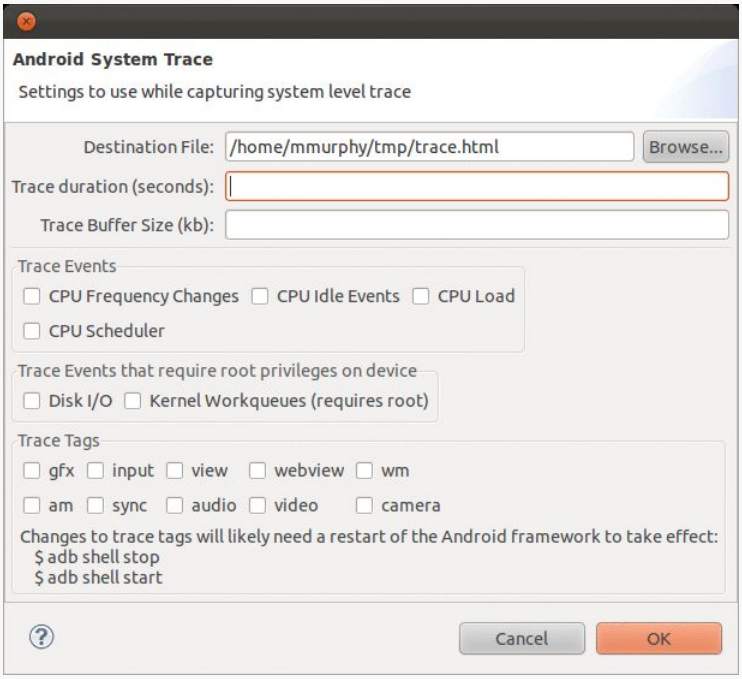


Figure 757: Systrace Dialog in Eclipse

Notable settings that you will wish to tailor include:

- Where the trace will be written (by default, as `trace.html` in your home or user directory)
- The duration of the trace
- Which trace tags you wish to use

Clicking OK will then initiate the trace collection, at which point you will want to go to your test device and run through your test scenario.

Choosing the Trace Tags

All of these instructions have been telling you to specify what `systrace` tags to collect when you collect the trace data. So, what should you collect?

The big four are:

- `sched` for CPU scheduling
- `gfx` for graphics
- `view` for widget rendering

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- `wm` for window management

Apps using `WebView` might consider the `webview` tag. There are a variety of other tags as well that you might find useful for one analysis or another. However, be careful not to request "everything but the kitchen sink", as it may make your reports difficult to interpret.

Also note that not all devices support all tags. `python systrace.py --list-categories` should tell you what is possible for your connected device.

Augmenting the Trace from Java

You can effectively add your own tag to the output in Java code, to flag key sections of application processing and see where they fall in the report's timeline. To do this:

- Add calls to `Trace.beginSection()` and `Trace.endSection()` in your API Level 18+ app. Here, `Trace` is `android.os.Trace`, and `beginSection()` takes a `String` parameter that you would like to have logged. Note that these calls can nest, so you can have one section inside of another, but `endSection()` closes the last-begun section. Hence, make sure that your `beginSection()` and `endSection()` calls match up, typically by using `try/finally` exception handling. Also, your `beginSection()` and `endSection()` calls must match up in terms of threads — you cannot begin a section on one thread and end it on another.
- Add the `--app` switch to name your application's package if you are running `systrace` from the command line. Or, in DDMS, choose your application in the "Enable Application Traces from" drop-down list.

Viewing and Interpreting the Results

What you get as output is an HTML file that can be viewed in the Chrome browser, though you will tend to want to use a development machine for this instead of, say, an Android tablet. That is because the navigation of the Web page is designed for use with a hardware QWERTY keyboard, which most Android devices lack.

You can find a sample trace from a Nexus 7 online, though note that the HTML is a bit large and may take a few seconds to download. Initially, you will see something like this:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

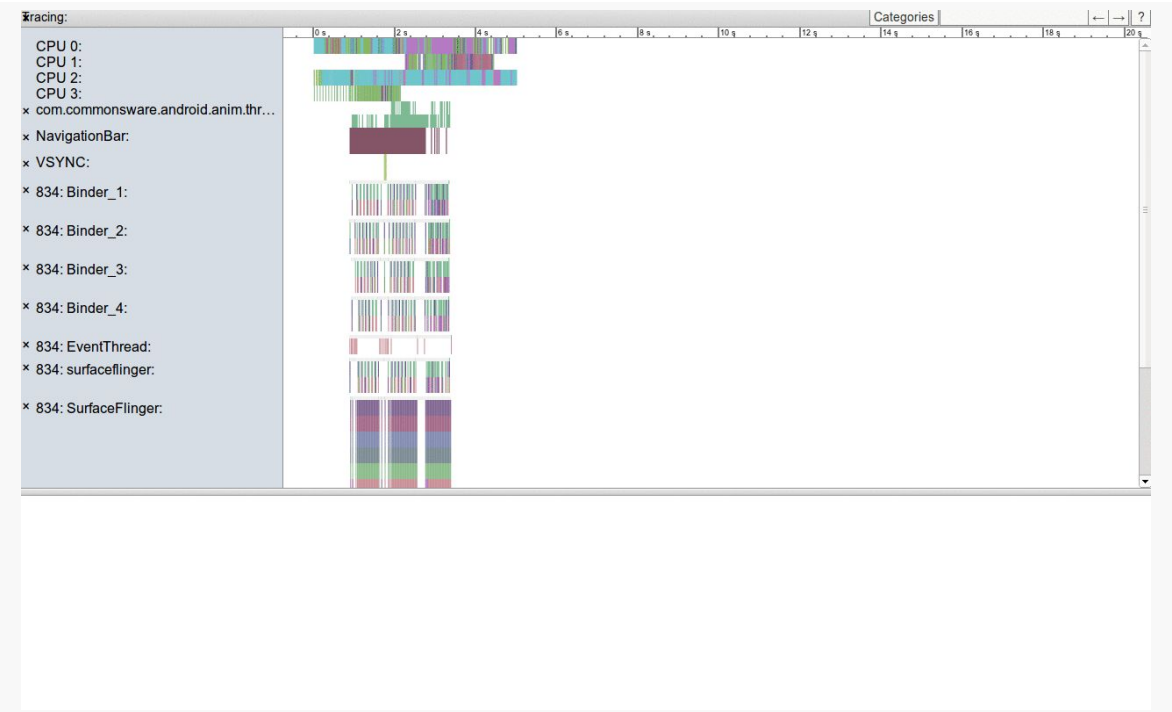


Figure 758: Systrace Output, As Initially Viewed

The left-hand sidebar represents various categories (or "slices" or "tags" or whatever) of data collected by `systrace`. The main area shows a timeline for the test, with rows corresponding to the sidebar entries for what was occurring at the various times for that particular category. The bottom pane will hold details that will appear when you click on various little blocks within that timeline.

Mostly, your navigation will use the W, A, S, and D keys, presumably chosen to make it appear as though you are playing a video game. Specifically:

- W will zoom in the timeline, while S will zoom out
- A and D will pane the timeline left and right

Jank will show up as gaps in the `SurfaceFlinger`:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

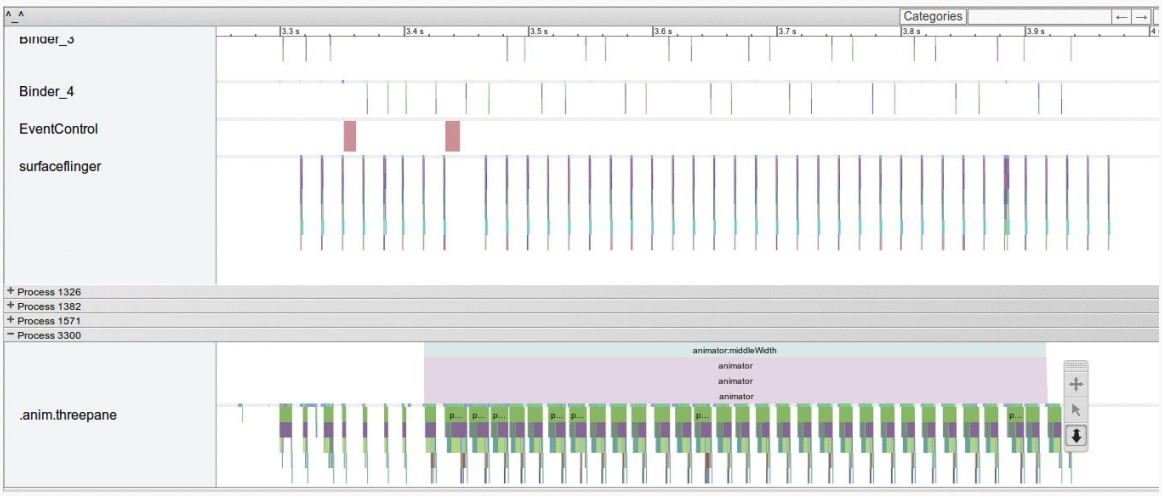


Figure 759: Systrace Output, Zoomed In on 0.7 Seconds of Profiling

Each of the major ticks across the timeline represents 0.1 seconds. There should be six frames in those seconds. However, we can see that in the 3.4–3.5 second range, there is a dropped frame, which shows up as a gap where there should be a pulse of `SurfaceFlinger` activity.

Zooming in further starts to bring up some detail for the threads in our process, showing methods within the view processing hierarchy that we were working on during this period of time:

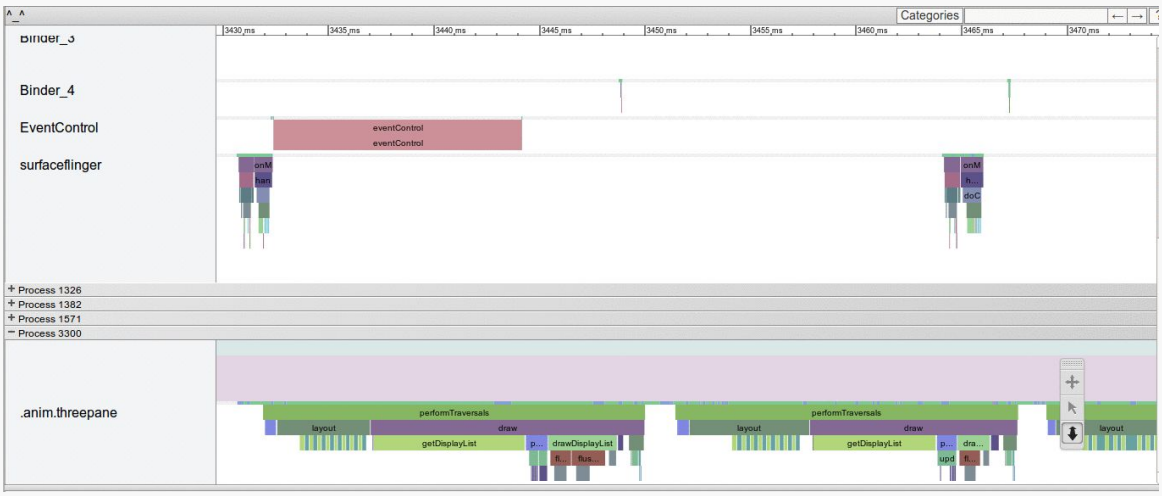


Figure 760: Systrace Output, Zoomed In on 40 Milliseconds of Profiling

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In our `gfx/view` slice, we will see various blocks for different major operations in the rendering of our UI. Notably, you will see blocks labeled "performTraversals", referring to the private `performTraversals()` method on `ViewRootImpl`. It turns out that `performTraversals()` wraps around all of the work shown in the three columns of our `gfxinfo` output: draw, process, and execute. The widths of the "performTraversals" blocks in the `systrace` output shows us how long each of those takes. What we want are nice, short blocks. Instead, panning through our trace, you will see several that are too long. The chapter on "jank busting" will go into further analysis of where this particular sample application went wrong that caused this behavior.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Focus On: NDK

When Android was first released, many a developer wanted to run C/C++ code on it. There was little support for this, other than by distributing a binary executable and running it via a forked process. While this works, it is a bit cumbersome, and the process-based interface limits how cleanly your C/C++ code could interact with a Java-based UI. On top of all of that, the use of such binary executables is not well supported.

In June 2009, the core Android team released the Native Development Kit (NDK). This allows developers to write C/C++ for Android applications in a supported fashion, in the form of libraries linked to a hosting Java-based application via the Java Native Interface (JNI). This offers a wealth of opportunities for Android development, and this part of the book will explore how you can take advantage of the NDK to exploit those opportunities.

This chapter explains how to set up the NDK and apply it to your project. What it does not do is attempt to cover all possible uses of the NDK — game applications in particular have access to many frameworks, like OpenGL and OpenSL, that are beyond the scope of this book.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate. Reading the introductory chapter to this trail is also a good idea.

This chapter also assumes that you know C/C++ programming.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Role of the NDK

We start by examining Dalvik's primary limitation — speed. Next, we look at the reasons one might choose the NDK, speed among them. We wrap up with some reasons why the NDK may not be the right solution for every Android problem, despite its benefits.

Dalvik: Secure, Yes; Speedy, Not So Much

Dalvik was written with security as a high priority. Android's security architecture is built around Linux's user model, with each application getting its own user ID. With each application's process running under its own user ID, one process cannot readily affect other processes, helping to contain any single security flaw in an Android application or subsystem. This requires a fair number of processes. However, phones have limited RAM, and the Android project wanted to offer Java-based development. Multiple processes hosting their own Java virtual machines simply could not fit in a phone. Dalvik's virtual machine is designed to address this, maximizing the amount of the virtual machine that can be shared securely between processes (e.g., via "copy-on-write").

Of course, it is wonderful that Android has security so woven into the fabric of its implementation. However, inventing a new virtual machine required tradeoffs, and most of those are related to speed.

A fair amount of work has gone into making Java fast. Standard Java virtual machines do a remarkable job of optimizing applications on the fly, such that Java applications can perform at speeds near their C/C++ counterparts. This borders on the amazing and is a testament to the many engineers who put countless years into Java.

Dalvik, by comparison, is very young. Many of Java's performance optimization techniques — such as advanced garbage collection algorithms — simply have not been implemented to nearly the same level in Dalvik. This is not to say they will never exist, but it will take some time. Even then, though, there may be limits as to how fast Dalvik can operate, considering that it cannot "throw memory at the problem" to the extent Java can on the desktop or server.

If you need speed, Dalvik is not the answer today, and may not be the answer tomorrow, either.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Going Native

Java-based Android development via Dalvik and the Android SDK is far and away the option with the best support from the core Android team. HTML5 application development is another option that was brought to you by the core Android development team. The third leg of the official Android development triad is the NDK, provided to developers to address some specific problems, outlined below.

Speed

Far and away the biggest reason for using the NDK is speed, pure and simple. Writing in C/C++ for the device's CPU will be a major speed improvement over writing the same algorithms in Java, despite Android's just-in-time (JIT) compiler.

There is overhead in reaching out to the C/C++ code from a hosting Java application, and so for the best performance, you will want a coarse interface, without a lot of calls back and forth between Java and the native opcodes. This may require some redesign of what might otherwise be the "natural" way of writing the C/C++ code, or you may just have to settle for less of a speed improvement. Regardless, for many types of algorithms — from cryptography to game AI to video format conversions — using C/C++ with the NDK will make your application perform much better, to the point where it can enable applications to be successful that would be entirely too slow if written solely in Java.

Bear in mind, though, that much of what you think is Java code in your app really is native "under the covers". Many of the built-in Android classes are thin shims over native implementations. Again, focus on applying the NDK where you are performing lots of work yourself in Java code that might benefit from the performance gains.

Porting

You may already have some C/C++ code, written for another environment, that you would like to use with Android. That might be for a desktop application. That might be for another mobile platform, such as iOS, where C/C++ is an option. That might be for mobile platform, such as Symbian, where C/C++ is the conventional solution, rather than some other language. Regardless, so long as that code is itself relatively platform-independent, it should be usable on Android.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This may significantly streamline your ability to support multiple platforms for your application, even if down-to-the-metal speed is not really something you necessarily need. This may also allow you to reuse existing C/C++ code written by others, for image processing or scripting languages or anything else.

Knowing Your Limits

Developers love silver bullets. Developers are forevermore seeking The One True Approach to development that will be problem-free. Sisyphus would approve, of course, as development always involves tradeoffs. So while the NDK's speed may make it tantalizing, it is not a solution for general Android application development, for several reasons, explored in this section.

Android APIs

The biggest issue with the NDK is that you have very limited access to Android itself. There are a few libraries bundled with Android that you can leverage, and a few other APIs offered specifically to the NDK, such as the ability to render OpenGL 3D graphics. But, generally speaking, the NDK has no access to the Android SDK, except by way of objects made available to it from the hosting application via JNI.

As such, it is best to view the NDK as a way of speeding up particular pieces of an SDK application — game physics, audio processing, OCR, and the like. All of those are algorithms that need to run on Android devices with data obtained from Android, but otherwise are independent of Android itself.

Cross-Platform Compatibility

While C/C++ *can* be written for cross-platform use, often it is not.

Sometimes, the disparity is one of APIs. Any time you use an API from a platform (e.g., iPhone) or a library (e.g., Qt) not available on Android, you introduce an incompatibility. This means that while a lot of your code — measured in terms of lines — may be fine for Android, there may be enough platform-specific bits woven throughout it that you would have a significant rewrite ahead of you to make it truly cross-platform.

Android itself, though, has a compatibility issue, in terms of CPUs. Android mostly runs on ARM devices today, since Android's initial focus was on smartphones, and ARM-powered smartphones at that. However, the focus on ARM will continue to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

waver, particularly as Android moves into other devices where other CPU architectures are more prevalent, such as Atom or MIPS for set-top boxes. While your code may be written in a fashion that works on all those architectures, the binaries that code produces will be specific to one architecture. The NDK gives you additional assistance in managing that, so that your application can simultaneously support multiple architectures.

Right now, the NDK supports ARM, x86, and MIPS CPU architectures. Of these, ARM CPUs power the *vast* majority of Android devices. The first generation of Google TV boxes, and a few other devices, use Intel x86 CPUs (usually Atom-based). MIPS is a relative newcomer to Android, with few devices using such CPUs at this time.

NDK Installation and Project Setup

The Android NDK is blissfully easy to install, in some ways even easier than is the Android SDK. Similarly, setting up an NDK-equipped project is rather straightforward. However, the documentation for the NDK is mostly a set of text files (`OVERVIEW.TXT` prominent among them). These are well-written but suffer from the limits of the plain-text form factor, plus are focused strictly on the NDK and not the larger issue of Android projects that use the NDK.

This chapter will fill in some of those gaps.

Installing the NDK

As with the Android SDK, the Android NDK comes in the form of a ZIP or `tar.gz` file, containing everything you need to build NDK-enabled Android applications. Hence, setting up the NDK is fairly trivial, particularly if you are developing on Linux.

Prerequisites

You will need the GNU `make` and GNU `awk` packages installed. These may be part of your environment already. For example, in Ubuntu, run `sudo apt-get install make gawk`, or use the Software Center, to ensure you have these two packages.

While you can do NDK development directly on Linux or OS X, NDK development on Windows can only be done using the Cygwin environment. This gives you a Linux-style shell and Linux-style tools on a Windows PC. In addition to a base

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Cygwin 1.7 (or newer) installation, you will need the `make` and `gawk` Cygwin packages installed in Cygwin.

If you encounter difficulties with Cygwin, you may wish to consider whether running Linux in a virtualization environment (e.g., VirtualBox) might be a better solution for you.

Download and Unpack

The Android NDK per-platform (Linux/OS X/Windows) ZIP files can be downloaded from the NDK page on the Android Developers site. These ZIP files are not small (~50MB each), because they contain the entire toolchain — that is why there are so few prerequisites.

You are welcome to unpack the ZIP file anywhere it makes sense on your development machine. However, putting it *inside* the Android SDK directory may not be a wise move — a peer directory would be a safer choice. You are welcome to rename the directory if you choose.

Environment Variables

The NDK documentation will cite an `NDK` environment variable, set to point to the directory in which you unpacked the NDK. This is a documentation convention and does not appear to be required for actual use of the NDK, though it is not a bad idea. You could also consider adding the NDK directory to your `PATH`, though that too is not required.

Bear in mind that you will be using the NDK tools from the command line, and so being able to conveniently reference this directory is reasonably important.

Setting Up an NDK Project

At its core, an NDK-enhanced Android project is a regular Android project. You still need a manifest, layouts, Java source code, and all the other trappings of a regular Android application. The NDK simply enables you to add C/C++ code to that project and have it included in your builds, referenced from your Java code via the Java Native Interface (JNI).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The examples shown in this section are from the JNI/WeakBench sample project, which implements a pair of benchmarks in Java and C, to help demonstrate the performance differences between the environments.

Writing Your C/C++ Code

The first step towards adding NDK code to your project is to create a `jni/` directory and place your C/C++ code inside of it. While there are ways to use a different base directory, it is unclear why you would need to. How you organize the code inside of `jni/` is up to you. C++ code should use `.cpp` as file extensions, though this too is configurable.

Your C/C++ code will be made up of two facets:

- The code doing the real work
- The code implementing your JNI interface

If you have never used JNI before, JNI uses naming conventions to tie functions in a C/C++ library to their corresponding hooks in the Java code.

For example, in the `WeakBench` project, you will find `jni/weakbench.c`:

```
#include <stdlib.h>
#include <math.h>
#include <jni.h>

typedef unsigned char boolean;

static void nsieve(int m) {
    unsigned int count = 0, i, j;
    boolean * flags = (boolean *) malloc(m * sizeof(boolean));
    memset(flags, 1, m);

    for (i = 2; i < m; ++i)
        if (flags[i]) {
            ++count;
            for (j = i << 1; j < m; j += i)
//                if (flags[j])
                  flags[j] = 0;
    }

    free(flags);
}

void
Java_com_commonsware_android_tuning_weakbench_WeakBench_nsievenative( JNIEnv*
env,
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                                                                    jobject
thiz )
{
    int i=0;
    for (i = 0; i < 3; i++)
        nsieve(10000 << (9-i));
}

double eval_A(int i, int j) { return 1.0/((i+j)*(i+j+1)/2+i+1); }

void eval_A_times_u(int N, const double u[], double Au[])
{
  int i,j;
  for(i=0;i<N;i++)
    {
      Au[i]=0;
      for(j=0;j<N;j++) Au[i]+=eval_A(i,j)*u[j];
    }
}

void eval_At_times_u(int N, const double u[], double Au[])
{
  int i,j;
  for(i=0;i<N;i++)
    {
      Au[i]=0;
      for(j=0;j<N;j++) Au[i]+=eval_A(j,i)*u[j];
    }
}

void eval_AtA_times_u(int N, const double u[], double AtAu[])
{ double v[N]; eval_A_times_u(N,u,v); eval_At_times_u(N,v,AtAu); }


void
Java_com_commonsware_android_tuning_weakbench_WeakBench_specnative( JNIEnv* env,
                                                                    jobject
thiz )
{
    int i;
    int N = 1000;
    double u[N],v[N],vBv,vv;
    for(i=0;i<N;i++) u[i]=1;
    for(i=0;i<10;i++)
      {
        eval_AtA_times_u(N,u,v);
        eval_AtA_times_u(N,v,u);
      }
    vBv=vv=0;
    for(i=0;i<N;i++) { vBv+=u[i]*v[i]; vv+=v[i]*v[i]; }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Much of the code shown here comes from the Great Language Benchmarks Game, specifically their `nsieve` and `spectral-norm` benchmarks. And, much of the code looks like normal C code.

Two functions, though, serve as JNI entry points:

- `Java_com_commonsware_abj_weakbench_WeakBench_nsievenative`
- `Java_com_commonsware_abj_weakbench_WeakBench_specnative`

As will be seen later in this section, these will map to `nsievenative()` and `specnative()` methods on a `com.commonsware.abj.weakbench.WeakBench` class. The Java class (with package) and method names are converted into a function call name, so JNI can identify the function at runtime.

The implementation of these methods do not make use of any Java objects, nor do they return anything — they just implement the benchmark.

Writing Your Makefile(s)

To tell the NDK tools how to build your code, you will need one or two makefiles.

Android.mk

This makefile will describe the "module" (library) that you are attempting to add to your Android project by way of the NDK. In it, you will specify the source files that should be compiled and linked into the module. This file, by default, resides in the root of your `jni/` directory.

For example, here is `jni/Android.mk` from the `WeakBench` project:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := weakbench
LOCAL_SRC_FILES := weakbench.c

include $(BUILD_SHARED_LIBRARY)
```

Here, we give the module a name (`weakbench`) and identify the source files that go into it (`weakbench.c`).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

It is possible for you to have multiple `Android.mk` files, in multiple subdirectories of `jni/`, to create multiple modules. There is an `ANDROID-MK.TXT` file in the NDK documentation directory that provides more detail on how you can configure complex scenarios like this one.

Application.mk

There is a separate, optional, makefile that you can have, `Application.mk`, in your `jni/` directory. This is where you can provide compile flags for the build process, which CPU architectures (ARM, x86, etc.) you wish to support, and so on. By default, if you do not have such a file, the NDK build tools will include all modules defined in your `Android.mk` file(s) in your project, compiled for a generic ARM target with software support for floating-point operations.

For basic NDK applications, skipping `Application.mk` is a reasonable choice. Complex projects, or ones specifically aiming to support other CPU architectures (e.g., ARM-v7 CPUs with hardware floating-point support), will need an `Application.mk` file.

The `WeakBench` project has a one-line `Application.mk` file:

```
APP_ABI := all
```

This tells Android that we want to build the JNI code for all supported CPU architectures. At the time of this writing, that is ARMv5, ARMv7, and x86.

Building Your Library

Any time you modify your C/C++ code, or the makefiles, you will need to build your NDK library. To do that, from a command prompt in your project's root directory, run the `ndk-build` script found in the NDK's root directory. In other words, if you set up an NDK environment variable to point to where you have the NDK installed, execute `$NDK/ndk-build` from your project root.

This will compile and link your C/C++ code into a library (or conceivably several libraries, if you have a complex set of `Android.mk` files). These will wind up in your project's `libs/` directory, in subdirectories based on your CPU architectures indicated by your `Application.mk` file.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

For example, if you run `$NDK/ndk-build` from the `WeakBench` project root, you will wind up with a `libs/armeabi/libweakbench.so` file. The `armeabi` portion is because that is the default CPU architecture that the NDK supports, and `WeakBench` did not change the defaults via an `Application.mk` file. The "weakbench" portion of `libweakbench.so` is because our `LOCAL_MODULE` value in our `Android.mk` file is `weakbench`. The `lib` prefix is automatically added by the build tools. The `.so` file extension is because our `Android.mk` file indicated that we are building a shared library (via the `BUILD_SHARED_LIBRARY` directive), and `.so` is the standard file extension for shared libraries in Linux (and, hence, Android).

Note that you will also wind up with similar `.so` files in `libs/armeabi-v7a/` and `libs/x86` for those architectures.

You are welcome to add this to your build process, such as adding it to your Ant build script, though it is not automatically included in the build process as defined by Android.

Using Your Library Via JNI

Now that you have your base C/C++ code being successfully compiled by the NDK, you need to turn your attention towards crafting the bridge between the Dalvik VM and the C/C++ code, following in the conventions of the Java Native Interface (JNI).

This section, while explaining the various steps involved in using the JNI, is far from a complete treatise on the subject. If you are going to spend a lot of time working with JNI, you are encouraged to seek additional resources on this topic, such as Core Java: Volume II, which has a chapter on JNI.

We created two C functions for accessing benchmarks:

- `Java_com_commonsware_abj_weakbench_WeakBench_nsievenative`
- `Java_com_commonsware_abj_weakbench_WeakBench_specnative`

Those, in turn, need to be defined as static methods on a `com.commonsware.abj.weakbench.WeakBench` class. Moreover, these methods will need to have the `native` keyword, indicating that their implementation is not found in Java code, but in native C/C++ code. The naming convention of the C functions allows the Dalvik runtime to identify what function names should be used for those `native` method implementations.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, that alone will be insufficient — we need to tell Dalvik where it can find the library in the first place. While naming conventions are good enough for the C function names, there is no corresponding naming convention for the library itself.

To do this, we use the `loadLibrary()` static method on the `System` class. A class implementing native methods should call `loadLibrary()` in a `static` block, so it is executed when the class is first referenced. For the NDK, all we need to do is supply the name we gave the library in the `Android.mk` file.

Here is the portion of the `WeakBench` class that has the `native` methods and the `loadLibrary()` call:

```
  static {
    System.loadLibrary("weakbench");
  }

  public native void nsievenative();
  public native void specnative();
```

Now, we can call our `nsievenative()` and `specnative()` methods on `WeakBench`, just as if they were regular Dalvik methods on a regular Dalvik class. The fact that they are really going off and invoking C functions is purely "implementation detail" that the consumers of those methods can be blissfully unaware of.

`WeakBench` itself is an `Activity`, invoking both Dalvik and native implementations of these two benchmarks. It uses a series of `AsyncTask` objects for executing the benchmarks on background threads, then updates `TextView` widgets in the UI to show the results:

```
package com.commonsware.android.tuning.weakbench;

import android.app.Activity;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.SystemClock;
import android.widget.TextView;

public class WeakBench extends Activity {
  static {
    System.loadLibrary("weakbench");
  }

  public native void nsievenative();
  public native void specnative();

  @Override
  public void onCreate(Bundle savedInstanceState) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    new JavaSieveTask().execute();
  }

  /*
   * Code after this point is adapted from the Great Computer Language
   * Shootout. Copyrights are owned by whoever contributed this stuff,
   * or possibly the Shootout itself, since there isn't much information
   * on ownership there. Licensed under a modified BSD license.
   */

  private class JavaSieveTask extends AsyncTask<Void, Void, Void> {
    long start=0;
    TextView result=null;

    @Override
    protected void onPreExecute() {
      result=(TextView)findViewById(R.id.nsieve_java);

      result.setText("running...");
    }

     @Override
    protected Void doInBackground(Void... unused) {
      start=SystemClock.uptimeMillis();

      int n=9;
      int m=(1<<n)*10000;
      boolean[] flags=new boolean[m+1];

      nsieve(m,flags);

      m=(1<<n-1)*10000;
      nsieve(m,flags);

      m=(1<<n-2)*10000;
      nsieve(m,flags);

      return(null);
    }

    @Override
    protected void onPostExecute(Void unused) {
      long delta=SystemClock.uptimeMillis()-start;

      result.setText(String.valueOf(delta));
      new JavaSpecTask().execute();
    }
  }

  private class JavaSpecTask extends AsyncTask<Void, Void, Void> {
    long start=0;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  TextView result=null;

  @Override
  protected void onPreExecute() {
    result=(TextView)findViewById(R.id.spec_java);

    result.setText("running...");
  }

   @Override
  protected Void doInBackground(Void... unused) {
    start=SystemClock.uptimeMillis();

    Approximate(1000);

    return(null);
  }

  @Override
  protected void onPostExecute(Void unused) {
    long delta=SystemClock.uptimeMillis()-start;

    result.setText(String.valueOf(delta));
    new JNISieveTask().execute();
  }
}

private class JNISieveTask extends AsyncTask<Void, Void, Void> {
  long start=0;
  TextView result=null;

  @Override
  protected void onPreExecute() {
    result=(TextView)findViewById(R.id.nsieve_jni);

    result.setText("running...");
  }

   @Override
  protected Void doInBackground(Void... unused) {
    start=SystemClock.uptimeMillis();

    nsievenative();

    return(null);
  }

  @Override
  protected void onPostExecute(Void unused) {
    long delta=SystemClock.uptimeMillis()-start;

    result.setText(String.valueOf(delta));
    new JNISpecTask().execute();
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
}

private class JNISpecTask extends AsyncTask<Void, Void, Void> {
  long start=0;
  TextView result=null;

  @Override
  protected void onPreExecute() {
    result=(TextView)findViewById(R.id.spec_jni);

    result.setText("running...");
  }

   @Override
  protected Void doInBackground(Void... unused) {
    start=SystemClock.uptimeMillis();

    specnative();

    return(null);
  }

  @Override
  protected void onPostExecute(Void unused) {
    long delta=SystemClock.uptimeMillis()-start;

    result.setText(String.valueOf(delta));
  }
}

private static int nsieve(int m, boolean[] isPrime) {
    for (int i=2; i <= m; i++) isPrime[i] = true;
    int count = 0;

    for (int i=2; i <= m; i++) {
       if (isPrime[i]) {
          for (int k=i+i; k <= m; k+=i) isPrime[k] = false;
          count++;
       }
    }
    return count;
}

private final double Approximate(int n) {
  // create unit vector
  double[] u = new double[n];
  for (int i=0; i<n; i++) u[i] =  1;

  // 20 steps of the power method
  double[] v = new double[n];
  for (int i=0; i<n; i++) v[i] = 0;

  for (int i=0; i<10; i++) {
    MultiplyAtAv(n,u,v);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      MultiplyAtAv(n,v,u);
    }

    // B=AtA         A multiplied by A transposed
    // v.Bv /(v.v)   eigenvalue of v
    double vBv = 0, vv = 0;
    for (int i=0; i<n; i++) {
      vBv += u[i]*v[i];
      vv  += v[i]*v[i];
    }

    return Math.sqrt(vBv/vv);
  }


  /* return element i,j of infinite matrix A */
  private final double A(int i, int j){
    return 1.0/((i+j)*(i+j+1)/2 +i+1);
  }

  /* multiply vector v by matrix A */
  private final void MultiplyAv(int n, double[] v, double[] Av){
    for (int i=0; i<n; i++){
      Av[i] = 0;
      for (int j=0; j<n; j++) Av[i] += A(i,j)*v[j];
    }
  }

  /* multiply vector v by matrix A transposed */
  private final void MultiplyAtv(int n, double[] v, double[] Atv){
    for (int i=0;i<n;i++){
      Atv[i] = 0;
      for (int j=0; j<n; j++) Atv[i] += A(j,i)*v[j];
    }
  }

  /* multiply vector v by matrix A and then by matrix A transposed */
  private final void MultiplyAtAv(int n, double[] v, double[] AtAv){
    double[] u = new double[n];
    MultiplyAv(n,v,u);
    MultiplyAtv(n,u,AtAv);
  }
}
```

As with our C implementations of the benchmarks, the Java source code is derived from the Great Language Benchmarks Game.

Building and Deploying Your Project

Given that you have done all of this, the rest is perfectly normal – you build and deploy your Android project no differently than if you did not have any C/C++ code.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Your native library is embedded in your APK file, so you do not have to worry about distributing it separately.

However, bear in mind that the more architectures you choose, the more `.so` files there are and the bigger your app will be. For tiny bits of C/C++ code, like the code in this app, this increase in file size will not be very noticeable. However, it is something to keep in mind for more elaborate NDK applications. Moreover, there are some ways to help reduce the impact of these extra architectures.

libhoudini and the NDK

`libhoudini` is a proprietary ARM translation layer for x86-powered Android devices. It allows an app that has NDK binaries for ARM, but not x86, to still run on x86 hardware, albeit not as quickly as it would with native x86 binaries.

Given ARM's current dominance in the Android ecosystem, `libhoudini` is hugely useful for Intel and hardware vendors interested in using Intel's mobile CPUs. Without it, only apps that ship x86 NDK binaries would be compatible with x86-powered devices like the Samsung Galaxy Tab 3 10.1" tablet. Some developers probably skip x86 NDK binaries, because they are not aware of popular x86-powered devices, or lack one for testing, or are concerned over APK size. The Play Store for x86 would shrink substantially from the million-plus apps available to ARM devices, to those that do not use the NDK or happen to ship x86 binaries. `libhoudini` makes ARM-only NDK binaries usable on x86, giving x86-powered Android devices access to more of the Play Store catalog.

However, it *is* slower. A test suite for SQLCipher for Android, run on an ASUS MeMO Pad FHD 10, ran about three times as long when using the ARM binaries and `libhoudini`, when compared to the same test run using x86 binaries. On the other hand, supporting x86 in addition to ARM adds another 5MB to the app, on top of the 6.5MB spent for ARM and the platform-neutral pieces. Being able to use SQLCipher for Android without the x86 binaries might be useful, particularly for apps bumping up against APK size limits, like the 50MB limit on the Play Store.

You may wish to do your own testing. Testing is easy enough: just temporarily move the `x86/` directory from `libs/` somewhere else, then recompile and test on `libhoudini`-equipped hardware. If you do not have your own `libhoudini`-equipped hardware, you may be able to take advantage of services like Samsung's Remote Test Lab, which recently added the Galaxy Tab 3 10.1 to its lineup.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you have the space for it, include the x86 binaries for your NDK-compiled libraries. This will give you maximum speed for little incremental engineering cost. However, if space is at a premium, `libhoudini` may allow you to reach many of the same x86 devices, but be sure that your app will run acceptably given the performance overhead.

Gradle and the NDK

Version 0.7 of the Gradle for Android plugin added preliminary support for the NDK. However, this support has its issues, and so you may wish to pursue other approaches, at least in the short term.

Official Support, for Externally-Built Binaries

Sometimes, you may have a project for which you want to use NDK-compiled binaries that somebody else supplies.

For example, at the time of this writing, SQLCipher for Android is not available as an AAR dependency from any repo. To use it, you need to include a handful of JARs in your project, along with NDK-compiled binaries for the core SQLCipher library and related libraries. Once this is available as an AAR, you could get all of that via a dependency; in the short term, you need to teach Gradle how to pick up the NDK-compiled binaries.

Using 0.7.+ (or higher) of the Gradle for Android plugin, you can have a `jniLibs/` directory in a sourceset. Underneath that directory would go your pre-compiled binaries, in the standard CPU architecture directories (e.g., `jniLibs/x86/`, `jniLibs/armeabi-v7a/`).

The `Gradle/ConstantsSecure` sample project is a clone of the same-named project from the SQLCipher for Android chapter, but with the code reorganized into a Gradle sourceset:

```
ConstantsSecure
|— build.gradle
|— libs
|    |— commons-codec.jar
|    |— guava-r09.jar
|    |— sqlcipher.jar
|— local.properties
|— proguard.cfg
|— project.properties
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
|── src
   |── main
      |── AndroidManifest.xml
      |── assets
      |    |── icudt46l.zip
      |── java
      |    |── com
      |        └── commonsware
      |            └── android
      |                └── sqlcipher
      |                    |── ConstantsBrowser.java
      |                    |── DatabaseHelper.java
      |                    └── Provider.java
      |── jniLibs
      |    |── armeabi
      |    |    |── libdatabase_sqlcipher.so
      |    |    |── libsqlcipher_android.so
      |    |    |── libstlport_shared.so
      |    |── x86
      |        |── libdatabase_sqlcipher.so
      |        |── libsqlcipher_android.so
      |        |── libstlport_shared.so
      └── res
         |── drawable-hdpi
         |    |── ic_launcher.png
         |── drawable-ldpi
         |    |── ic_launcher.png
         |── drawable-mdpi
         |    |── add.png
         |    |── cw.png
         |    |── delete.png
         |    |── eject.png
         |    |── ic_launcher.png
         |── drawable-xhdpi
         |    |── ic_launcher.png
         |── layout
         |    |── add_edit.xml
         |    |── main.xml
         |    |── row.xml
         └── values
           |── strings.xml
```

(note: above listing includes only files of relevance for the current discussion)

The JARs remain in `libs/`, but the associated NDK `.so` files go in `jniLibs/` of the `main` sourceset.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

When organized this way, the `build.gradle` file needs no changes to incorporate the `.so` files. If you wanted to have the JNI `.so` files in some other directory, you can modify `jniLibs.srcDirs` of your sourceset to point to where you want the files to reside.

Official Support, for Building NDK Binaries

To use the official NDK support for invoking the NDK build process from your `build.gradle` file, you need to have a `local.properties` file, defining an `ndk.dir` property, pointing to where your NDK is installed.

You can add an `ndk` closure to your `defaultConfig`, describing how your NDK code can be built:

```
defaultConfig {
    ndk {
        moduleName "anddown"
    }
}
```

In addition to `moduleName`, you can specify:

- `cFlags` for compiler flags
- `ldLibs` for libraries to link in

Your `productFlavors` can also have `ndk` closures with `abiFilter` properties, identifying a particular set of NDK binaries to be included in that flavor. For example, here is a `build.gradle` sample file from a test project for the Gradle for Android plugin:

```
buildscript {
    repositories {
        maven { url '../../../../out/host/gradle/repo' }
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.7.0-SNAPSHOT'
    }
}
apply plugin: 'android'

android {
    compileSdkVersion 15
    buildToolsVersion "18.0.1"

    defaultConfig {
        ndk {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
            moduleName "sanangeles"
            cFlags "-DANDROID_NDK -DDISABLE_IMPORTGL"
            ldLibs "GLESv1_CM", "dl", "log"
            stl "stlport_static"
        }
    }

    buildTypes.debug.jniDebugBuild true

    productFlavors {
        x86 {
            ndk {
                abiFilter "x86"
            }
        }
        arm {
            ndk {
                abiFilter "armeabi-v7a"
            }
        }
        mips {
            ndk {
                abiFilter "mips"
            }
        }
    }
}
```

Note that this sample is seriously out of date, using Gradle for Android 0.7.0, but it helps to illustrate how the product flavors work.

In here, we define three product flavors, one each for x86, ARM, and MIPS. Each resulting binary will only contain that architecture's version of the compiled NDK code.

Also, you need to tell Gradle for Android where your C/C++ source code resides. The default location is a `jni/` directory in your sourcesets, such as the `main/` sourceset. However, you can override that using `jni.srcDirs` in your configuration of a sourceset.

Example: CWAC-AndDown

CWAC-AndDown is mentioned in passing in the chapter on rich text handling. It is an Android library that wraps hoedown, a C-based Markdown-to-HTML converter. The hoedown project itself is a fork of sundown, which itself was used by many sites, like GitHub, for their Markdown processing. CWAC-AndDown is great for projects

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

that take in Markdown and want to render the results in a `WebView` (though you can use `Html.fromHtml()` to put the results in a `TextView` if you want).

Since CWAC-AndDown uses hoedown's C code, it has a `jni/` directory containing the JNI wrapper C code, the appropriate makefiles, plus a `src/` subdirectory containing the hoedown source code.

The classic way to build the project was to manually run the `ndk-build` script in the project root, then use Ant to compile the demo. CWAC-AndDown itself is an Android library project, designed to be used by apps like the demo app.

To get this library project to build using Ant and Eclipse, in addition to Gradle, we needed to leave the `jni/` directory in its original spot, alongside the `res/`, `src/`, and related directories and files. Since that is not where Gradle for Android goes looking for C/C++ files, we needed to teach Gradle for Android where our JNI code resides, by configuring `jni.srcDirs` for our `main` `sourceset` in `build.gradle`:

```
apply plugin: 'maven'

uploadArchives {
    repositories.mavenDeployer {
        pom.groupId = PUBLISH_GROUP_ID
        pom.artifactId = PUBLISH_ARTIFACT_ID
        pom.version = PUBLISH_VERSION

        repository(url: LOCAL_REPO)
    }
}
```

The CWAC-AndDown `build.gradle` file also contains the `ndk` closure shown earlier in this section, to name the module that we are generating.

Unofficial Support for Makefiles

We also needed to reorganize the CWAC-AndDown source code, due to some fundamental limitations in the current official Android support. Specifically, Gradle for Android *ignores* your own makefiles, preferring instead to generate its own. The generation algorithms are rather limited and make some assumptions about the organization of your code. Having all of your C/C++ source and header files in one big directory seems to work; having other structures may or may not work.

The fact that Gradle for Android ignores your makefiles means that if your makefiles do anything beyond very vanilla stuff, your build may well not work.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The alternative to using the built-in NDK support is to add in your own tasks that use the `ndk-build` command with your original makefiles. One such set of tasks was published by David Weinstein in a GitHub Gist. However, this recipe was written for an early beta version of the Gradle for Android plugin and therefore may require some modification.

Mostly, it involves adding the following to the bottom of your `build.gradle` file:

```
task packageNativeLibs_ARM(type: Jar) {
    baseName 'libtestlib'
    classifier 'armeabi'
    from(file('libs/armeabi/')) {
        include '**/*.so'
    }
    into('lib/armeabi')
    destinationDir(file('libs/'))
}

task packageNativeLibs_x86(type: Jar) {
    baseName 'libtestlib'
    classifier 'x86'
    from(file('libs/x86/')) {
        include '**/*.so'
    }
    into('lib/x86')
    destinationDir(file('libs/'))
}

task packageNativeLibs(description: "package native libraries") {

}

packageNativeLibs.dependsOn 'packageNativeLibs_ARM'
packageNativeLibs.dependsOn 'packageNativeLibs_x86'

task ndkBuild(type: Exec, description: "Task to run ndk-build") {
    commandLine ndkDir + '/ndk-build'
}

packageNativeLibs.dependsOn 'ndkBuild'

tasks.withType(JavaCompile) { compileTask -> compileTask.dependsOn
packageNativeLibs }

clean.dependsOn 'cleanPackageNativeLibs'
```

This task also assumes that you have set an `ndkDir` property. To set properties in Gradle, add a `gradle.properties` file to your project, using the typical properties format:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
ndkDir=/opt/android-ndk
```

The Gradle code snippet does several things:

1. It defines two tasks, one each to package the ARM and x86 version of the binaries. If you have a different mix of NDK build targets (e.g., ARMv7, MIPS), you could add or alter these tasks to suit.
2. It defines a `packageNativeLibs` task, that depends upon each of the CPU architecture-specific tasks from the previous point. If you add, change, or remove those architecture-specific tasks, you will need to add, change, or remove the corresponding `packageNativeLibs.dependsUpon` statements.
3. It defines an `ndkBuild` task, that simply executes the `ndk-build` script found in your NDK directory (defined by that `ndkDir` property).
4. It has `packageNativeLibs` depend upon `ndkBuild`.
5. It has all `JavaCompile` tasks in the build depend upon the `packageNativeLibs` task. This ensures that when we compile our Java code, we also compile the NDK code, if it is not already up to date.
6. It adds the auto-generated `cleanPackageNativeLibs` task as a dependency for the `clean` task, so `gradle clean` will clean the NDK build as well as cleaning everything else.

Gradle supports other ways of defining where the NDK directory is, such as depending upon an environment variable, so you have alternatives.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Improving CPU Performance in Java

Knowing that you have CPU-related issues in your app is one thing — doing something about it is the next challenge. In some respects, tuning an Android application is a "one-off" job, tied to the particulars of the application and what it is trying to accomplish. That being said, this chapter will outline some general-purpose ways of boosting performance that may counter issues that you are running into.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate. Reading the introductory chapter to this trail is also a good idea.

Reduce CPU Utilization

One class of CPU-related problems come from purely sluggish code. These are the sorts of things you will see in Traceview, for example – methods or branches of code that seem to take an inordinately long time. These are also some of the most difficult to have general solutions for, as often times it comes down to what the application is trying to accomplish. However, the following sections provide suggestions for consuming fewer CPU instructions while getting the same work done.

These are presented in no particular order.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Standard Java Optimizations

Most of your algorithm fixes will be standard Java optimizations, no different than have been used by Java projects over the past decade and change. This section outlines a few of them. For more, consider reading *Effective Java* by Joshua Bloch or *Java Performance Tuning* by Jack Shirazi.

Avoid Excessive Synchronization

Few objects in `java.*` namespaces are intrinsically thread-safe, outside of `java.util.concurrent`. Typically, you need to perform your own synchronization if multiple threads will be accessing non-thread-safe objects. However, sometimes, Java classes have synchronization that you neither expect nor need. Synchronization adds unnecessary overhead.

The classic example here is `StringBuffer` and `StringBuilder`. `StringBuffer` was part of Java from early on, and, for whatever reason, was written to be thread-safe — two threads that append to the buffer will not cause any problems. However, most of the time, you are only using the `StringBuffer` from one thread, meaning all that synchronization overhead is a waste. Later on, Java added `StringBuilder`, with the same basic set of methods as has `StringBuffer`, but without the synchronization.

Similarly, in your own code, only synchronize where it is really needed. Do not toss the `synchronized` keyword around randomly, or use concurrent collections that will only be used by one thread, etc.

Avoid Floating-Point Math

The first generation of Android devices lacked a floating-point coprocessor on the ARM CPU package. As a result, floating-point math speed was atrocious. That is why the Google Maps add-on for Android uses `GeoPoint`, with latitude and longitude in integer microdegrees, rather than the standard Android `Location` class, which uses Java double variables holding decimal degrees.

While later Android devices do have floating-point coprocessor support, that does not mean that floating-point math is now as fast as integer math. If you find that your code is spending lots of time on floating-point calculations, consider whether a change in units would allow you to replace the floating-point calculations with integer equivalents. For example, microdegrees for latitude and longitude provide

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

adequate granularity for most maps, yet allow Google Maps to do all of its calculations in integers.

Similarly, consider whether the full decimal accuracy of floating-point values is really needed. While it may be physically possible to perform distance calculations in meters with accuracy to a few decimal points, for example, in many cases the user will not need that degree of accuracy. If so, perhaps changing to fixed-point (integer) math can boost your performance.

Don't Assume Built-In Algorithms are Best

Years upon years of work has gone into the implementation of various algorithms that underlie Java methods, like searching for substrings inside of strings.

Somewhat less work has gone into the implementation of the Apache Harmony versions of those methods, simply because the project is younger, and it is a modified version of the Harmony implementation that you will find in Android. While the core Android team has made many improvements to the original Harmony implementation, those improvements may be for optimizations that do not fit your needs (e.g., optimizing to reduce memory consumption at the expense of CPU time).

But beyond that, there are dozens of string-matching algorithms, some of which may be better for you depending on the string being searched and the string being searched for. Hence, you may wish to consider applying your own searching algorithm rather than relying on the built-in one, to boost performance. And, this same concept may hold for other algorithms as well (e.g., sorting).

Of course, this will also increase the complexity of your application, with long-term impacts in terms of maintenance cost. Hence, do not assume the built-in algorithms are the worst, either — optimize those algorithms that Traceview or logging suggest are where you are spending too much time.

Support Hardware-Accelerated Graphics

An easy "win" is to add `android:hardwareAccelerated="true"` to your `<application>` element in the manifest. This toggles on hardware acceleration for 2D graphics, including much of the stock widget framework. For maximum backwards compatibility, this hardware acceleration is off, but adding the aforementioned attribute will enable it for all activities in your application.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that this is only available starting with Android 3.0. It is safe to have the attribute in the manifest for older Android devices, as they simply will ignore your request.

You also should test your application thoroughly after enabling hardware acceleration, to make sure there are no unexpected issues. For ordinary widget-based applications, you should encounter no problems. Games or other applications that do their own drawing might have issues. If you find that some of your code runs into problems, you can override hardware acceleration on a per-activity basis by putting the `android:hardwareAccelerated` on `<activity>` elements in the manifest.

Minimize IPC

Calling a method on an object in your own process is fairly inexpensive. The overhead of the method invocation is fairly minuscule, and so the time involved is simply however long it takes for that method to do its work.

Invoking behaviors in another process, via inter-process communication (IPC), is considerably more expensive. Your request has to be converted into a byte array (e.g., via the `Parcelable` interface), made available to the other process, converted back into a regular request, then executed. This adds substantial CPU overhead.

There are three basic flavors of IPC in Android:

1. "Directly" invoking a third-party application's service's AIDL-published interface, to which you bound with `bindService()`
2. Performing operations on a content provider that is not part of your application (i.e., supplied by the OS or a third-party application)
3. Performing other operations that, under the covers, trigger IPC

Remote Bound Service

Using a remote service is fairly obvious when you do it — it is difficult to mistake copying the AIDL into your project and such. The proxy object generated from the AIDL converts all your method calls on the interface into IPC operations, and this is relatively expensive.

If you are exposing a service via AIDL, design your API to be coarse-grained. Do not require the client to make 1,000 method invocations to accomplish something that can be done in 1 via slightly more complex arguments and return values.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you are consuming a remote service, try not to get into situations where you have to make lots of calls in a tight loop, or per row of a scrolled `AdapterView`, or anything else where the overhead may become troublesome.

For example, in the `CPU-Java/AIDLOverhead` sample project, you will find a pair of projects implementing the same do-nothing method in equivalent services. One uses AIDL and is bound to remotely from a separate client application; the other is a local service in the client application itself. The client then calls the do-nothing method 1 million times for each of the two services. On average, on a Samsung Galaxy Tab 10.1, 1 million calls takes around 170 seconds for the remote service, while it takes around 170 *milliseconds* for the local service. Hence, the overhead of an individual remote method invocation is small (~170 microseconds), but doing lots of them in a loop, or as the user flings a `ListView`, might become noticeable.

Remote Content Provider

Using a content provider can be somewhat less obvious of a problem. Using `ContentResolver` or a `CursorLoader` looks the same whether it is your own content provider or someone else's. However, you know what content providers you wrote; anything else is probably running in another process.

As with remote services, try to aggregate operations with remote content providers, such as:

1. Use `bulkInsert()` rather than lots of individual `insert()` calls
2. Try to avoid calling `update()` or `delete()` in a tight loop – instead, if the content provider supports it, use a more complex "WHERE clause" to update or delete everything at once
3. Try to get all your data back in few queries, rather than lots of little ones... though this can then cause you issues in terms of memory consumption

Remote OS Operation

The content provider scenario is really a subset of the broader case where you request that Android do something for you and winds up performing IPC as part of that.

Sometimes, this is going to be obvious. If you are sending commands to a third-party service via `startService()`, by definition, this will involve IPC, since the third-party

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

service will run in a third-party process. Try to avoid calling `startService()` lots of times in close succession.

However, there are plenty of cases that are less obvious:

1. All requests to `startActivity()`, `startService()`, and `sendBroadcast()` involve IPC, as it is a separate OS process that does the real work
2. Registering and unregistering a `BroadcastReceiver` (e.g., `registerReceiver()`) involves IPC
3. All of the "system services", such as `LocationManager`, are really rich interfaces to an AIDL-defined remote service, and so most operations on these system services require IPC

Once again, your objective should be to minimize calls that involve IPC, particularly where you are making those calls frequently in close succession, such as in a loop. For example, frequently calling `getLastKnownLocation()` will be expensive, as that involves IPC to a system process.

Android-Specific Java Optimizations

The way that the Dalvik VM was implemented and operates is subtly different than a traditional Java VM. Therefore, there are some optimizations that are more important on Android than you might find in regular desktop or server Java.

The Android developer documentation has a roster of such optimizations. Some of the highlights include:

1. Getters and setters, while perhaps useful for encapsulation, are significantly slower than direct field access. For simpler cases, such as `ViewHolder` objects for optimizing an Adapter, consider skipping the accessor methods and just use the fields directly.
2. Some popular method calls are replaced by hand-created assembler instructions rather than code generated via the JIT compiler. `indexOf()` on `String` and `arraycopy()` on `System` are two cited examples. These will run much faster than anything you might create yourself in Java.

Reduce Time on the Main Application Thread

Another class of CPU-related problem is when your code may be efficient, but it is occurring on the main application thread, causing your UI to react sluggishly. You

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

might have tuned your decryption algorithm as best as is mathematically possible, but it may be that decrypting data on the main application thread simply takes too much time. Or, perhaps `StrictMode` complained about some disk or network I/O that you are performing on the main application thread.

The following sections recap some commonly-seen patterns for moving work off the main application thread, plus a few newer options that you may have missed.

Generate Less Garbage

Most developers think of having too many allocations as being solely an issue of heap space. That certainly has an impact, and depending on the nature of the allocations (e.g., bitmaps), it may be the dominant issue.

However, garbage has impacts from a CPU standpoint as well. Every object you create causes its constructor to be executed. Every object that is garbage-collected requires CPU time both to find the object in the heap and to actually clean it up (e.g., execute the finalizer, if any).

Worse still, on older versions of Android (e.g., Android 2.2 and down), the garbage collector interrupts the entire process to do its work, so the more garbage you generate, the more times you "stop the world". Game developers have had to deal with this since Android's inception. To maintain a 60 FPS refresh rate, you cannot afford *any* garbage collections on older devices, as a single GC run could easily take more than the ~16ms you have per drawing pass.

As a result of all of this, game developers have had to carefully manage their own object pools, pre-allocating a bunch of objects before game play begins, then using and recycling those objects themselves, only allowing them to become garbage after game play ends.

Most non-game Android applications may not have to go to quite that extreme across the board. However, there are cases where excessive allocation may cause you difficulty. For example, avoiding creating too much garbage is one aspect of view recycling with `AdapterView`, which is covered in greater detail in the next section.

If Traceview indicates that you are spending a lot of time in garbage collection, pay attention to your loops or things that may be invoked many times in rapid succession (e.g., accessing data from a custom `Cursor` implementation that is tied to a `CursorAdapter`). These are the most likely places where your own code might be creating lots of extra objects that are not needed. Examining the heap to see what is

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

all being created (and eventually garbage collected) will be covered in an upcoming chapter of the book.

View Recycling

Perhaps the best-covered Android-specific optimization is view recycling with `AdapterView`.

In a nutshell, if you are extending `BaseAdapter`, or if you are overriding `getView()` in another adapter, please make use of the `View` parameter supplied to `getView()` (referred to here as `convertView`). If `convertView` is not `null`, it is one of your previous `View` objects you returned from `getView()` before, being offered to you for recycling purposes. Using `convertView` saves you from inflating or manually constructing a fresh `View` every time the user scrolls, and both of those operations are relatively expensive.

If you have been ignoring `convertView` because you have more than one type of `View` that `getView()` returns, your `Adapter` should be overriding `getViewTypeCount()` and `getItemViewType()`. These will allow Android to maintain separate object pools for each type of row from your `Adapter`, so `getView()` is guaranteed to be passed a `convertView` that matches the row type you are trying to create.

A somewhat more advanced optimization — caching all those `findViewById()` lookups — is also possible once your row recycling is in place. Often referred to as "the holder pattern", you do the `findViewById()` calls when you inflate a new row, then attach the `findViewById()` results to the row itself via some custom "holder" object and the `setTag()` method on `View`. When you recycle the row, you can get your "holder" back via `getTag()` and skip having to do the `findViewById()` calls again.

Background Threads

Of course, the backbone of any strategy to move work off the main application thread is to use background threads, in one form or fashion. You will want to apply these in places where `StrictMode` complains about network or disk I/O, or places where Traceview or logging indicate that you are taking too much time on the main application thread during GUI processing (e.g., converting downloaded bitmap images into `Bitmap` objects via `BitmapFactory`).

Sometimes, you will manually dictate where work should be done in the background, either by forking threads yourself or by using `AsyncTask`. `AsyncTask` is a

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

nice framework, handling all of the inter-thread communication for you and neatly packaging up the work to be done in readily understood methods. However, `AsyncTask` does not fit every scenario — it is mostly designed for "transactional" work that is known to take a modest amount of time (milliseconds to seconds) then end. For cases where you need unbounded background processing, such as monitoring a socket for incoming data, forking your own thread will be the better approach.

Sometimes, you will use facilities supplied by Android to move work to the background. For example, many activities are backed by a `Cursor` obtained from a database or content provider. Classically, you would manage the cursor (via `startManagingCursor()`) or otherwise arrange to refresh that `Cursor` in `onResume()`, so when your activity returns to the foreground after having been gone for a while, you would have fresh data. However, this pattern tends to lead to database I/O on the main application thread, triggering complaints from `StrictMode`. Android 3.0 and the Android Compatibility Library offer a `Loader` framework designed to try to solve the core pattern of refreshing the data, while arranging for the work to be done asynchronously.

Asynchronous BroadcastReceiver Operations

99.44% of the time (approximately) that Android calls your code in some sort of event handler, you are being called on the main application thread. This includes manifest-registered `BroadcastReceiver` components — `onReceive()` is called on the main application thread. So any work you do in `onReceive()` ties up that thread (possibly impacting an activity of yours in the foreground), and if you take more than 10 seconds, Android will terminate your `BroadcastReceiver` with extreme prejudice.

Classically, manifest-registered `BroadcastReceiver` components only live as long as the `onReceive()` call does, meaning you can do very little work in the `BroadcastReceiver` itself. The typical pattern is to have it send a command to a service via `startService()`, where the service "does the heavy lifting".

Android 3.0 added a `goAsync()` method on `BroadcastReceiver` that can help a bit here. While under-documented, it tells Android that you need more time to complete the broadcast work, but that you can do that work on a background thread. This does not eliminate the 10-second rule, but it does mean that the `BroadcastReceiver` can do some amount of I/O without having to send a command to a service to do it while still not tying up the main application thread.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The CPU-Java/GoAsync sample project demonstrates goAsync() in use, as the project name might suggest.

Our activity's layout consists of two Button widgets and an EditText widget:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_width="match_parent"
  android:layout_height="match_parent">
  <EditText android:id="@+id/editText1" android:layout_width="match_parent"
    android:layout_height="wrap_content">
  </EditText>
  <Button android:layout_width="match_parent" android:id="@+id/button1"
    android:layout_height="wrap_content" android:text="@string/nonasync"
    android:onClick="sendNonAsync"></Button>
  <Button android:layout_width="match_parent" android:id="@+id/button2"
    android:layout_height="wrap_content" android:text="@string/async"
    android:onClick="sendAsync"></Button>
</LinearLayout>
```

The activity itself simply has sendAsync() and sendNonAsync() methods, each invoking sendBroadcast() to a different BroadcastReceiver implementation:

```
package com.commonsware.android.tuning.goasync;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class GoAsyncActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void sendAsync(View v) {
    sendBroadcast(new Intent(this, AsyncReceiver.class));
  }

  public void sendNonAsync(View v) {
    sendBroadcast(new Intent(this, NonAsyncReceiver.class));
  }
}
```

The NonAsyncReceiver simulates doing time-consuming work in onReceive() itself:

```
package com.commonsware.android.tuning.goasync;

import android.content.BroadcastReceiver;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;

public class NonAsyncReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context arg0, Intent arg1) {
    SystemClock.sleep(7000);
  }
}
```

Hence, if you click the "Send Non-Async Broadcast" button, not only will the button fail to return to its normal state for seven seconds, but the `EditText` will not respond to user input either.

The `AsyncReceiver`, though, uses `goAsync()`:

```
package com.commonsware.android.tuning.goasync;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;

public class AsyncReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    final BroadcastReceiver.PendingResult result=goAsync();

    (new Thread() {
      public void run() {
        SystemClock.sleep(7000);
        result.finish();
      }
    }).start();
  }
}
```

The `goAsync()` method returns a `PendingResult`, which supports a series of methods that you might ordinarily fire on the `BroadcastReceiver` itself (e.g., `abortBroadcast()`) but want to do on a background thread. You need your background thread to have access to the `PendingResult` — in this case, via a `final` local variable. When you are done with your work, call `finish()` on the `PendingResult`.

If you click the "Send Async Broadcast" button, even though we are still sleeping for 7 seconds, we are doing so on a background thread, and so our user interface is still responsive.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Saving SharedPreferences

The classic way to save `SharedPreferences.Editor` changes was via a call to `commit()`. This writes the preference information to an XML file on whatever thread you are on — another hidden source of disk I/O you might be doing on the main application thread.

If you are on API Level 9, and you are willing to blindly try saving the changes, use the new `apply()` method on `SharedPreferences.Editor`, which works asynchronously.

If you need to support older versions of Android, or you really want the boolean return value from `commit()`, consider doing the `commit()` call in an `AsyncTask` or background thread.

And, of course, to support both of these, you will need to employ tricks like conditional class loading. You can see that used for saving `SharedPreferences` in the CPU-Java/PrefsPersist sample project. The activity reads in a preference, puts the current value on the screen, then updates the preference with the help of an `AbstractPrefsPersistStrategy` class and its `persist()` method:

```
package com.commonsware.android.tuning.prefs;

import android.app.Activity;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.widget.TextView;

public class PrefsPersistActivity extends Activity {
  private static final String KEY="counter";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    SharedPreferences prefs=
      PreferenceManager.getDefaultSharedPreferences(this);
    int counter=prefs.getInt(KEY, 0);

    ((TextView)findViewById(R.id.value)).setText(String.valueOf(counter));

    AbstractPrefsPersistStrategy.persist(prefs.edit().putInt(KEY, counter+1));
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`AbstractPrefsPersistStrategy` is an abstract base class that will hold a strategy implementation, depending on Android version. On pre-Honeycomb builds, it uses an implementation that forks a background thread to perform the `commit()`:

```
package com.commonsware.android.tuning.prefs;

import android.content.SharedPreferences;
import android.os.Build;

abstract public class AbstractPrefsPersistStrategy {
  abstract void persistAsync(SharedPreferences.Editor editor);

  private static final AbstractPrefsPersistStrategy INSTANCE=initImpl();

  public static void persist(SharedPreferences.Editor editor) {
    INSTANCE.persistAsync(editor);
  }

  private static AbstractPrefsPersistStrategy initImpl() {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk<Build.VERSION_CODES.HONEYCOMB) {
      return(new CommitAsyncStrategy());
    }

    return(new ApplyStrategy());
  }

  static class CommitAsyncStrategy extends AbstractPrefsPersistStrategy {
    @Override
    void persistAsync(final SharedPreferences.Editor editor) {
      (new Thread() {
        @Override
        public void run() {
          editor.commit();
        }
      }).start();
    }
  }
}
```

On Honeycomb and higher, it uses a separate strategy class that uses the new `apply()` method:

```
package com.commonsware.android.tuning.prefs;

import android.content.SharedPreferences.Editor;

public class ApplyStrategy extends AbstractPrefsPersistStrategy {

  @Override
  void persistAsync(Editor editor) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    editor.apply();
  }
}
```

By separating the Honeycomb-specific code out into a separate class, we can avoid loading it on older devices and encountering the dreaded `VerifyError`.

Whether using the built-in `apply()` method is worth dealing with multiple strategies, versus simply calling `commit()` on a background thread, is up to you.

Improve Throughput and Responsiveness

Being efficient and doing work on the proper thread may still not be enough. It could be that your work is not consuming excessive CPU time, but is taking too long in "wall clock time" (e.g., the user sits waiting too long at a `ProgressDialog`). Or, it could be that your work, while efficient and in the background, is causing difficulty for foreground operations.

The following sections outline some common problems and solutions in this area.

Minimize Disk Writes

Earlier in this book, we emphasized moving disk writes off to background threads.

Even better is to get rid of some of the disk writes entirely.

A big culprit here comes in the form of database operations. By default, each `insert()`, `update()`, or `delete()`, or any `execSQL()` invocation that modifies data, will occur in its own transaction. Each transaction involves a set of disk writes. Many times, this is not a problem. But, if you are doing a lot of these – such as importing records from a CSV file — hundreds or thousands of transactions will mean thousands of individual disk writes, and that can take some time. You may wish to wrap those operations in your own transaction, using methods like `beginTransaction()`, simply to reduce the number of transactions and, therefore, disk writes.

If you are doing your own disk I/O beyond databases, you may encounter similar sorts of issues. Overall, it is better to do a few larger writes than lots of little ones.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Set Thread Priority

Threads you fork, by default, run at a default priority: `THREAD_PRIORITY_DEFAULT` as defined on the `Process` class. This is a lower priority than the main application thread (`THREAD_PRIORITY_DISPLAY`).

Threads you use via `AsyncTask` run at a lower priority (`THREAD_PRIORITY_BACKGROUND`). If you fork your own threads, then, you might wish to consider moving them to a lower priority as well, to affect how much time they get compared to the main application thread. You can do this via `setThreadPriority()` on the `Process` class.

The lowest possible priority, `THREAD_PRIORITY_LOWEST`, is described as "only for those who really, really don't want to run if anything else is happening". You might use this for "idle-time processing", but bear in mind that the thread will be paused a lot to allow other threads to run.

Lower-priority threads will help ensure that your background work does not affect your foreground UI. Processes themselves are put in a lower-priority class as they move to the background (e.g., you have no activities visible), which further reduces the amount of CPU time you will be using at any given moment.

Also, note that `IntentService` uses a thread at default (*not* background) priority — you may wish to drop the priority of this thread to something that will be lower than your main application thread, to minimize how much CPU time the `IntentService` steals from your UI.

Do the Work Some Other Time

Just because you could do the work now does not mean you should do the work now. Perhaps a better answer is to do the work later, or do part of the work now and part of the work later.

For example, suppose that you have your own database of points of interest for your custom map application. Periodically, you publish a new database on your Web site, which your Android app should download. Odds are decent that the user is not in desperate need for this new database right away. In fact, the CPU time and disk I/O time to download and save the database might incrementally interfere with the foreground application, despite your best efforts.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In this case, not only should you check for and download the database when the user is unlikely to be using the device (e.g., before dawn), but you should check whether the screen is on via `isScreenOn()` on `PowerManager`, and delay the work to sometime when the screen is off. For example, you could have `AlarmManager` set up to have your code check for updates every 24 hours at 4am. If, at 4am, the screen is on, your code could skip the download and wait until tomorrow, or skip the download and add a one-shot alarm to wake you up in 30 minutes, in hopes that the user will no longer be using the device.

At the same time, you may wish to consider having a "refresh" menu choice somewhere, for when the user specifically wants you to go get the update (if available) *now*, for whatever reason.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Finding and Eliminating Jank

A user interface is considered "janky" if it stutters or otherwise fails to operate smoothly, particularly during animated effects like scrolling. Finding and eliminating the causes of janky behavior ("jank") is part science, part art, and part throwing darts at a dartboard.

This chapter will outline some techniques for identifying and removing jank from a user interface. The steps shown here originated in a blog post by Google's Romain Guy, with a few additional twists and turns due to the different nature of the particular case being studied. Mr. Guy's blog post is essential reading for all advanced Android developers, and the author is deeply indebted to Mr. Guy for his work in this area.

Prerequisites

The only hard prerequisite for this chapter is having read the core chapters and the chapter on finding CPU bottlenecks.

That being said, having read the chapter on animators would help understand portions of this chapter a bit better.

The Case: ThreePaneDemoBC

In the chapter on animators, we examined an implementation of the Gmail-style three-pane layout with animated transitions (a.k.a., "The Three-Fragment Problem"). The implementation shown there originated with a Stack Overflow question with the solution presented in this book offered as an answer.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A commenter on that answer pointed out that he detected some stutter, even on decent hardware.

This chapter reviews the steps that were taken to determine if we really are doing things incorrectly, what specifically we are doing wrong, and what can be done to fix it.

Are We Janky?

In the eyes of this book's author, the three-pane implementation presented in the chapter on animators was perfectly reasonable on good hardware.

There are two lessons to take from this:

1. It is better to come up with an objective definition for "jank" and test to see if your code meets that definition at various points
2. The author of this book is very tolerant of janky user interfaces

The results shown in the chapter on CPU measurement for the `gfxinfo` and `systrace` tools comes from the three-pane demo code. The `gfxinfo` and the `systrace` results both point to the three-pane demo spending too much time doing work and therefore dropping some number of frames. This lines up with the visual report, and indicates that we have some work to do to try to improve matters.

Finding the Source of the Jank

Just because we know that we are janky does not mean that we have any idea what to do about it. We need to conduct some further analysis to determine where, exactly, our jank is coming from.

Traceview

One thing that we can do to help further refine the source of our trouble is to use Traceview. As outlined in the section on Traceview, Traceview reports how many calls were made of various methods in our code (and in the framework code) and how much time was spent there.

Here are some of the results from a Traceview run on the three-pane demo on a Nexus 7:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

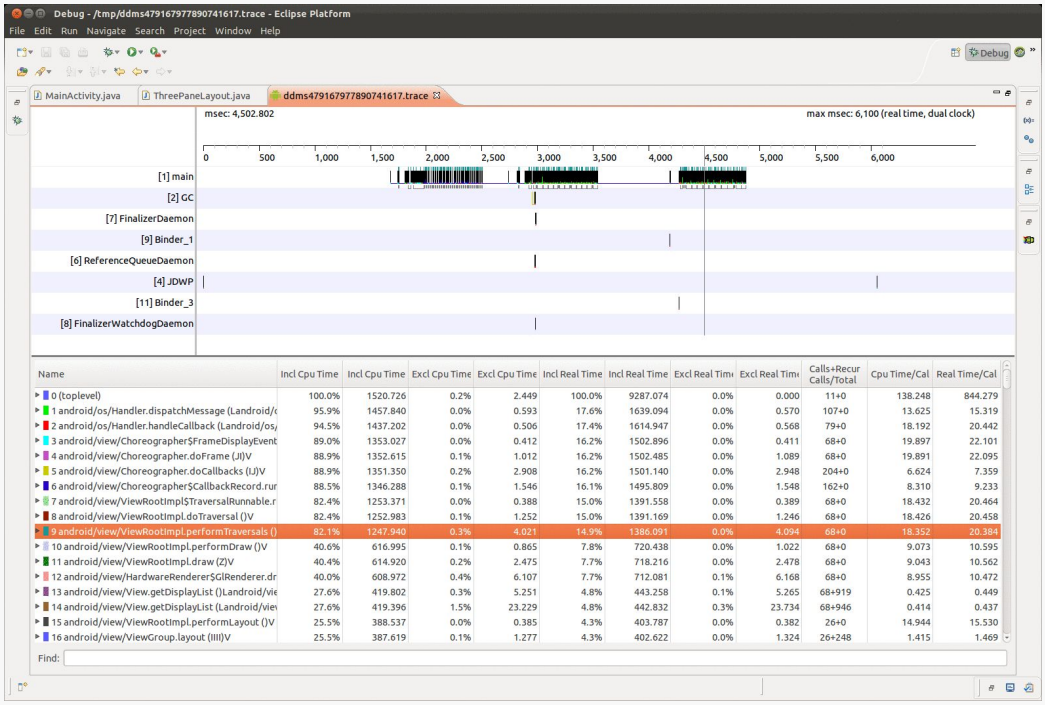


Figure 761: Traceview of Three-Pane Demo

We see that 88.9% of our CPU time is spent in `doFrame()` on `Choreographer` and the calls triggered from it. `doFrame()` is a private method which, as the name suggests, performs the drawing, processing, and executing of a single frame's worth of rendering. More importantly, we see that `doFrame()` was called 68 times during our test run, meaning that our UI changed 68 times during the ~3 seconds of activity during our trace.

Further down the table, we see that `layout()` on `ViewGroup` was called 26 times directly (and 248 more times via recursion), contributing about 25% of the time consumed by `doFrame()`. Since `layout()` is called on less than half of the `doFrame()` calls, the time consumed by `layout()` makes up a fairly significant portion of the `doFrame()` time during those 26 frames.

More importantly, `layout()` is something that *we* trigger. It implies that we have made some change to our UI content that requires a layout pass of some `ViewGroup`.

Having a layout pass on occasion is perfectly normal, particularly in response to user input. A `layout()` might be triggered by the user tapping on a row in one of our `ListViews`, for example. But we are not doing 26 user input events in our test — all we are doing is tapping one time each on a pair of `ListView` rows, then pressing the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

BACK button. This implies that something else in our code is causing `layout()` to be needed.

Unfortunately, at this point, Traceview does not help much, because the calls to `layout()` are asynchronous with respect to our own code, so it will not be all that obvious where the extra calls are coming from. This is where we need some expert help, as we will see later in this chapter.

Overdraw

Another common source of jank is overdraw. Overdraw refers to the act of painting the same pixel several times, due to overlapping components. For example:

- The activity window itself has a background
- You have a container that fills the activity's content, such as a `ListView`, which has a background
- You have children in that container with backgrounds (row), who have their own children with backgrounds and, eventually, content (widgets like `ImageView` and `TextView`)

Places where there is overlap, the OS might set the color of a pixel several times per frame, wasting time.

The easiest way to track down overdraw is to use the "Show GPU overdraw" option in the Developer Options portion of the Settings app:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

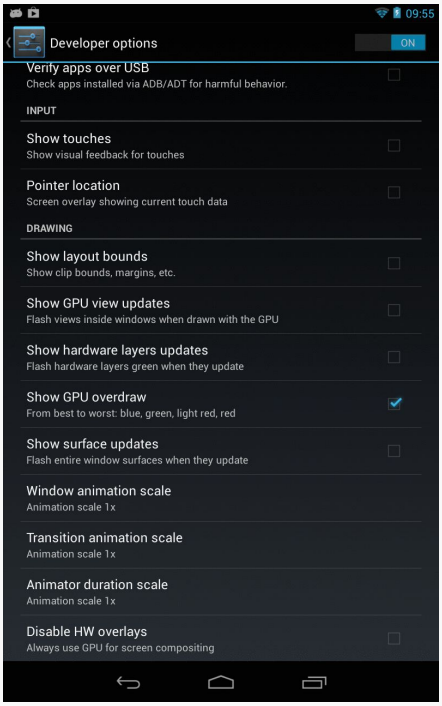


Figure 762: Nexus 7 Developer Options, with "Show GPU overdraw"

This option is only available on Android 4.2 and higher.

When you enable this option, then restart your app's process (if it was already running), Android will shade pixels that are overdrawn:

- Blue for pixels that are drawn twice
- Green for pixels that are drawn three times
- Pink for pixels that are drawn four times
- Red for pixels that are drawn five or more times

In short: pink and red are bad. Green and blue are OK, though if you have large patches of either shade, you might consider trying to see if there's a way to get rid of the overdraw.

Of course, the fact that these are shades applied to existing pixel colors may make it a bit difficult to tell exactly where the overdraw is occurring. For example, a red portion of your UI might be red from overdraw... or it might be red because you made it red. Temporarily changing your color scheme to something else (e.g., yellow) will help distinguish what is overdraw and what is just the natural UI coloration.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you enable this option on a Nexus 7 and run the three-pane demo, you will see very little blue or green (beyond the normal blue of the activated state of our `ListView` rows), and virtually no red:

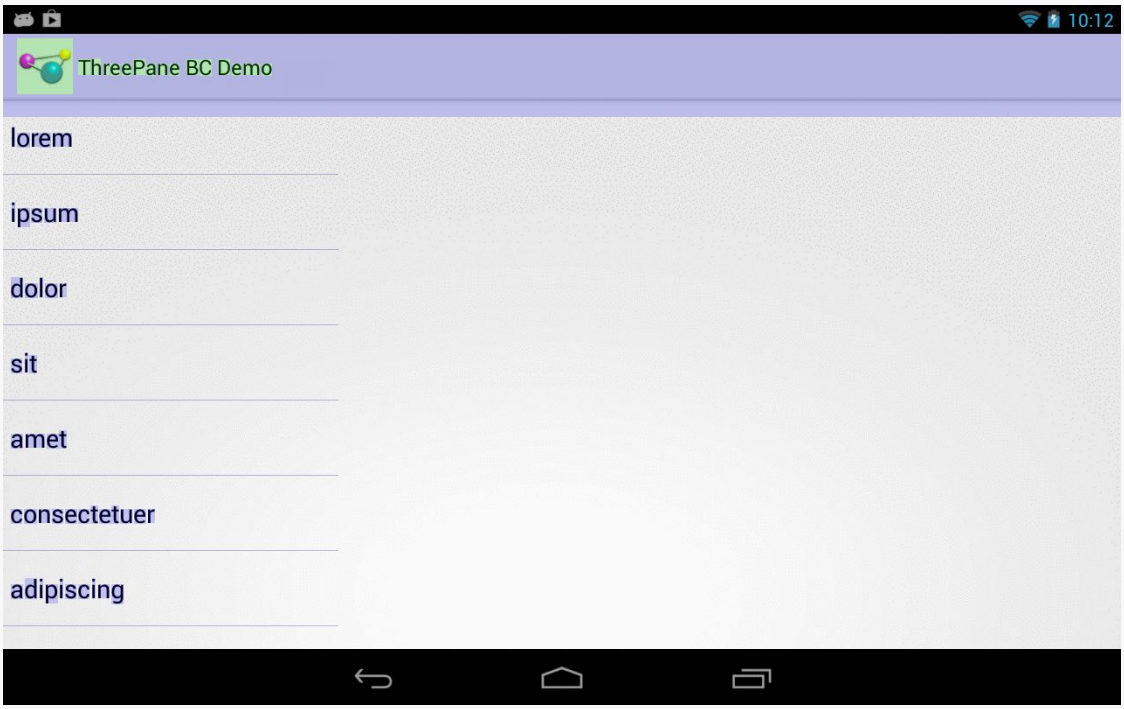


Figure 763: Three-Pane Demo, As Initially Launched, Showing Overdraw

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

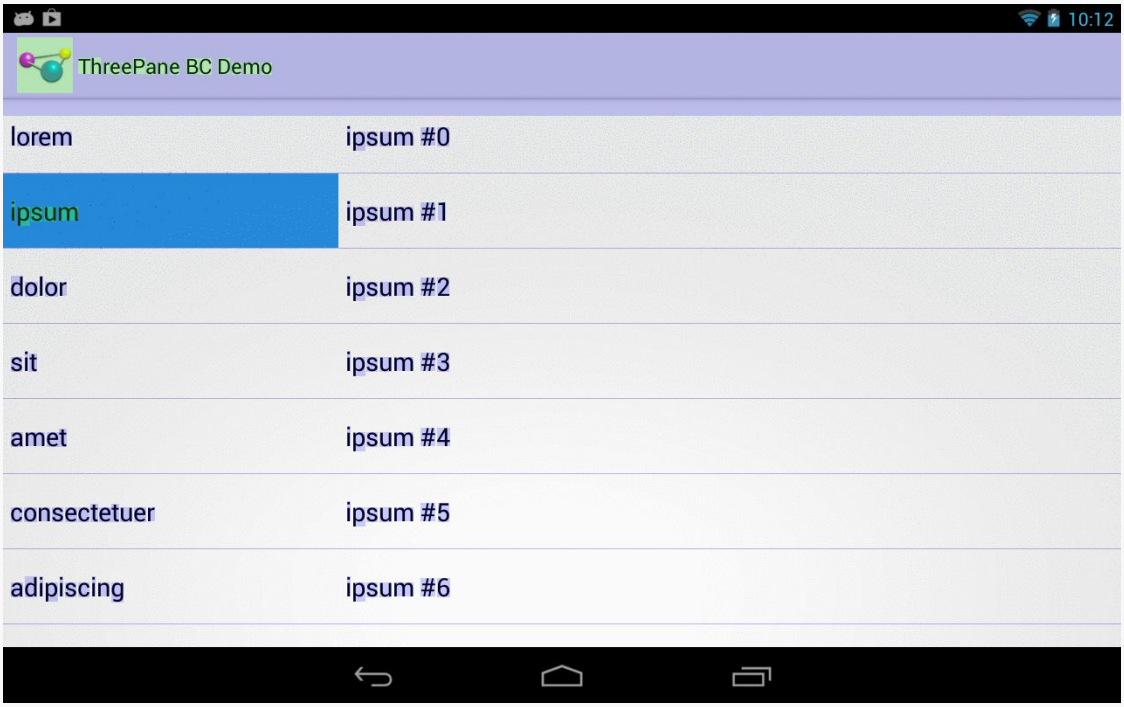


Figure 764: Three-Pane Demo, Left and Middle Panes, Showing Overdraw

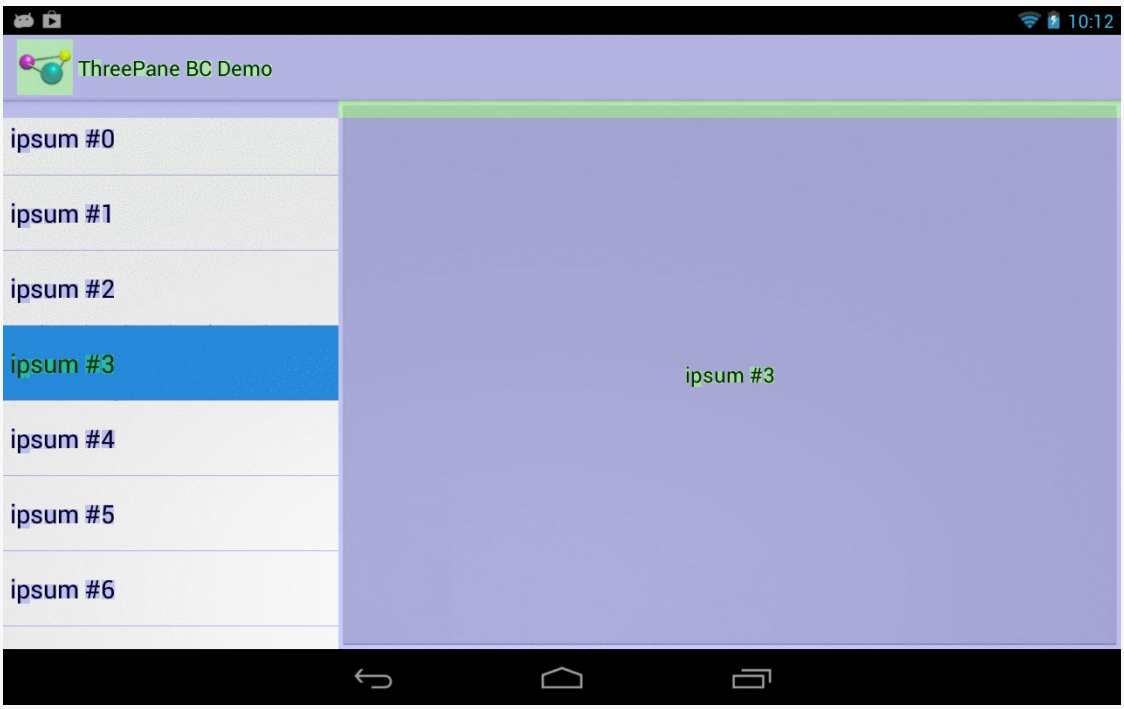


Figure 765: Three-Pane Demo, Middle and Right Panes, Showing Overdraw

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 766: Three-Pane Demo, Left and Middle Panes Via BACK, Showing Overdraw

On the other hand, bringing up the Contacts app on the same Nexus 7 shows significantly more overdraw:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

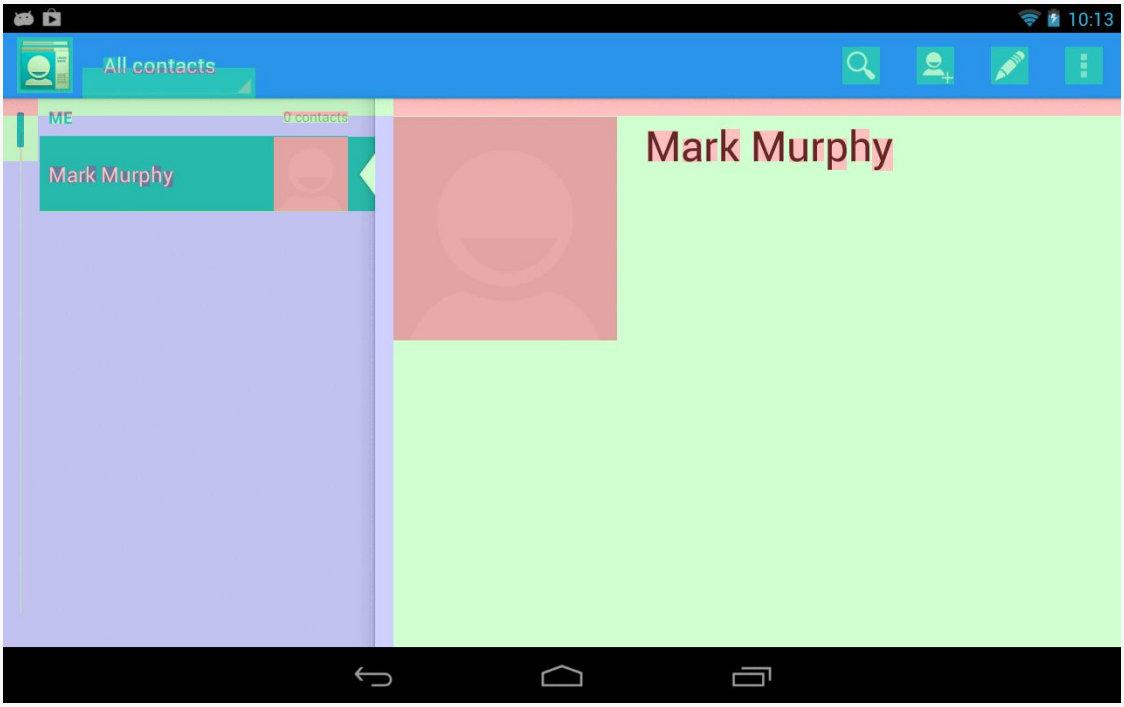


Figure 767: Contacts App, Showing Overdraw

The good news is that our app is not suffering performance problems due to overdraw.

The bad news is that the Contacts app is.

The good news is that if you are reading this, you are probably not responsible for maintaining the Contacts app.

The Contacts app's major problems come from the contact photos, or placeholders as seen here. Either the `ImageView` has a background, or the `ImageView` fills some container with a background. For example, the `ImageView` might be in some container with a background to provide a bevel effect around the image. Making the portion of the background that is behind the `ImageView` be transparent will eliminate the overdraw.

Note: some GPU architectures can automatically fix overdraw in select places, while others cannot. Notably, the Tegra 3 cannot. Hence, the Tegra 3 is a good test platform for using this overdraw-detection feature of Android.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Extraneous Views

Another related source of jank is having too many extraneous views. Each widget and container contributes to the cost of drawing the overall UI, so having extraneous views adds overhead.

Perhaps the most common scenario for extraneous views is the single-child container. If a container will only ever hold one child, perhaps you can get rid of that container. Not only will this speed up execution at runtime, but it can help avoid running out of stack space.

One likely way to find these extraneous views is to bring up your user interface in Hierarchy View on an emulator (or possibly on a device by using `ViewServer`). In particular, single-child containers are fairly obvious — look for bubbles that have just one child bubble on the right.

There are two such cases in the UI for our three-pane demo, though neither are our fault.

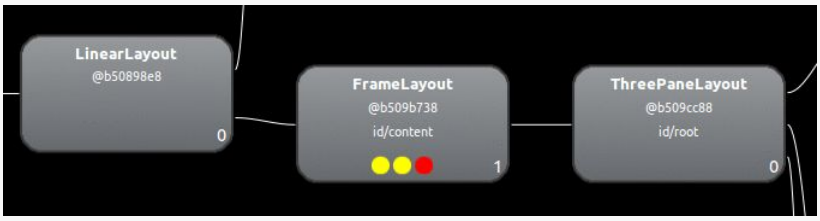


Figure 768: Single-Child FrameLayout in Three-Pane Demo, from Hierarchy View

Here, we have a `FrameLayout` holding onto just one child, our `ThreePaneLayout` custom view. We set up `ThreePaneLayout` as being our activity's content view. The "content view" of an activity is poured into a `FrameLayout`, supplied by the Android framework — that is the `FrameLayout` seen in Hierarchy View. We have no good way to get rid of this `FrameLayout`. Fortunately, `FrameLayout` is a very cheap container, in terms of runtime execution speed.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

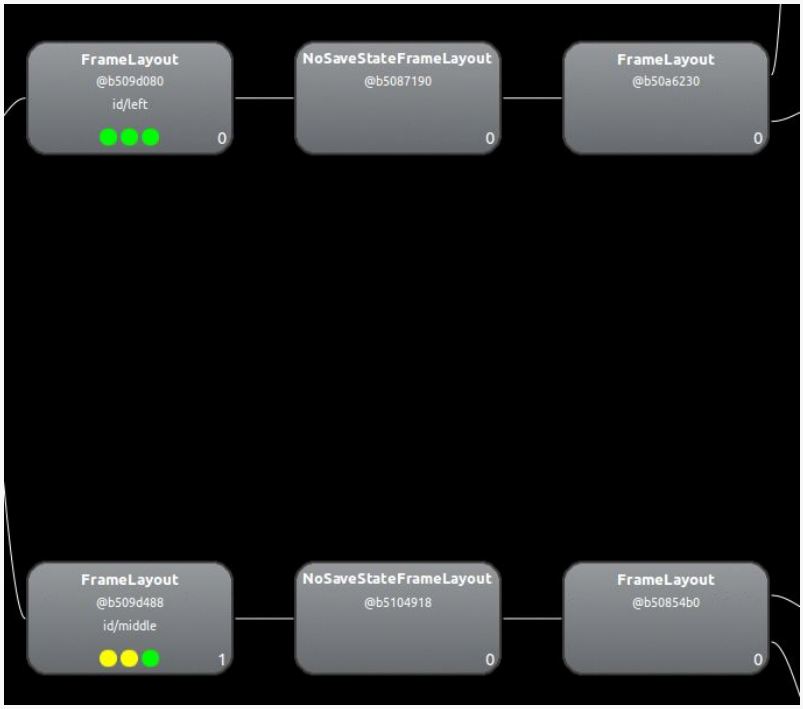


Figure 769: More Single-Child Containers in Three-Pane Demo, from Hierarchy View

Here, we see that our left and middle `FrameLayout` containers, for our left and middle panes, each contain one child, a `NoSaveStateFrameLayout`, which in turn each hold one child, a `FrameLayout`. These containers are added by `ListFragment`, not directly by our code. A `ListFragment` is surprisingly complex, adding several widgets and containers beyond the `ListView` itself:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

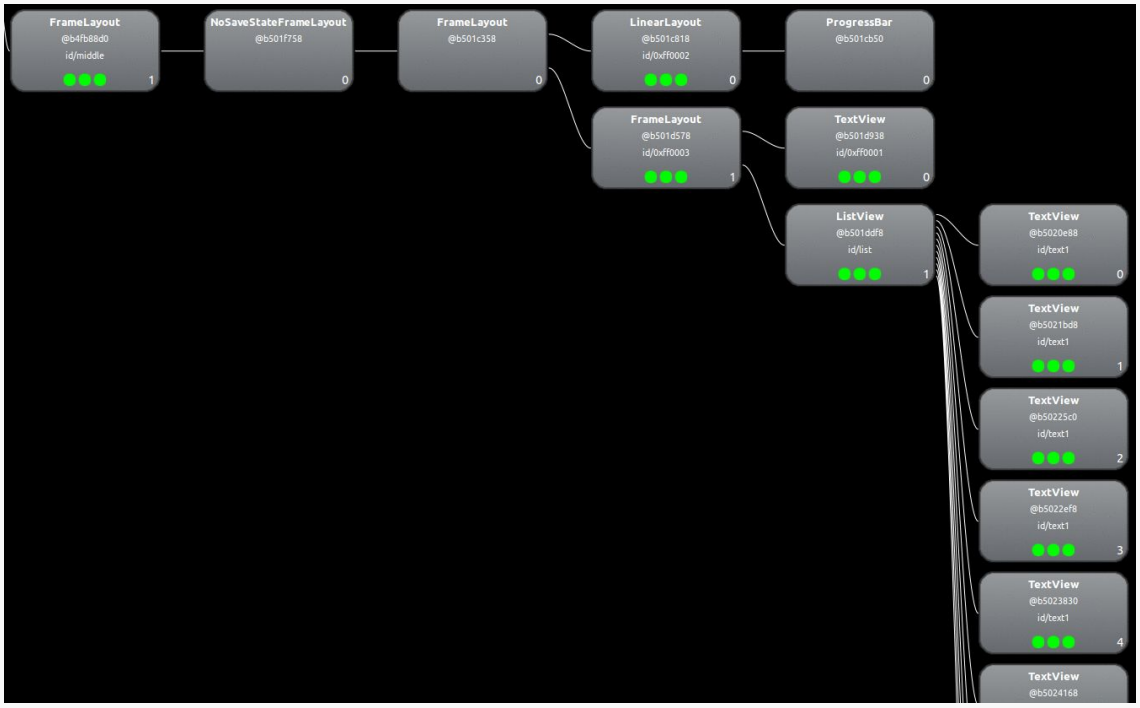


Figure 770: Contents of a ListFragment, from Hierarchy View

Short of writing our own fragment for holding a `ListView`, there is nothing we can do about these extraneous views.

Conclusion: Too Many layout() Calls?

Given that overdraw does not seem to be a problem and that we have few extraneous views under our control, it would seem that perhaps we should return our attention to the extra `layout()` calls. While trying to get rid of the `ListFragment` extraneous views would make those `layout()` calls incrementally cheaper, we will get more value by getting rid of the unnecessary calls in the first place, if indeed they are unnecessary.

Where Things Went Wrong

Of course, it doesn't hurt to call in an expert, to try to confirm exactly what is going on.

Chet Haase — Google engineer on Android, celebrated book author, and part-time comedian – chimed in with an answer to a Stack Overflow question about this three-

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

pane animation, asked by the person who commented about the dropped frames on the original Stack Overflow question.

The key statement from his answer was:

> Sliding things around is fine (translationX/Y), fading things in/out is good (alpha), but actually laying things out on every frame? Just say no.

Specifically, he is referring to our use of `ObjectAnimator` to change the width of the middle pane as we show and hide the right pane. Each time we change the width of the middle pane, we trigger a `layout()` call, to reposition the child widgets within that pane as needed. Our animations are adding ~20 `layout()` calls, introducing overhead that is pushing us over the per-frame limit on the Nexus 7.

Removing the Jank

To remove the jank, we need to remove the `ObjectAnimator` changing the width of the middle pane on the fly. You can see the results of this in the Jank/ThreePaneBC sample app.

Now, our `showLeft()` and `hideLeft()` methods immediately change the width of the middle pane, rather than arranging its animation:

```
public void hideLeft() {
  if (leftWidth == -1) {
    leftWidth=left.getWidth();
    middleWidthNormal=middle.getWidth();
    resetWidget(left, leftWidth);
    resetWidget(middle, middleWidthNormal);
    resetWidget(right, middleWidthNormal);
    requestLayout();
  }

  translateWidgets(-1 * leftWidth, left, middle, right);
  setMiddleWidth(leftWidth);
}

public void showLeft() {
  translateWidgets(leftWidth, left, middle, right);
  setMiddleWidth(middleWidthNormal);
}

private void setMiddleWidth(int value) {
  middle.getLayoutParams().width=value;
  requestLayout();
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This does not provide nearly as good of a UI as the original. However, the revised solution does reduce the jank, as seen in this `gfxinfo` output:

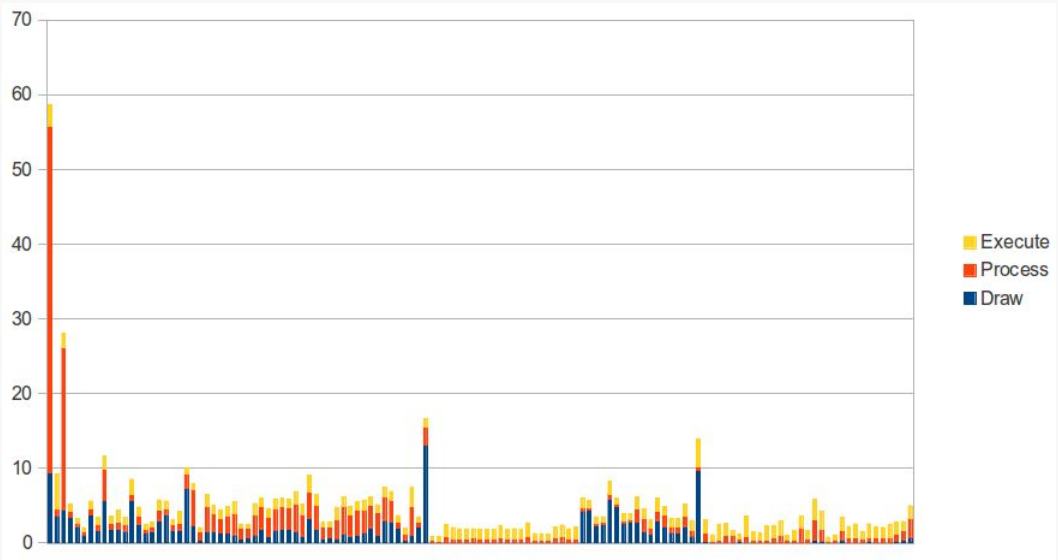


Figure 771: gfxinfo Output of Revised ThreePaneDemoBC

There are probably ways to improve upon the revised jank-free implementation. Lacking that, it is up to you to decide if the amount of jank found in the original implementation is worth the improved animation or not.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Issues with Bandwidth

As anyone who owned an Apple Newton or Palm V PDA back in the 1990's knows, handheld devices have been around for quite some time. For a very long time, they were a niche product, associated with geeks, nerds, and the occasional business executive.

Internet access changed all of that.

Blackberry for enterprise messaging — an outgrowth of its original two-way paging approach — blazed part of the trail, but the concept "crossed the chasm" to ordinary people with the advent of the iPhone, Android devices, and similar equipment.

Therefore, it is not terribly surprising when Android developers want to add Internet capabilities to their apps. To the contrary, it is almost unusual when you encounter an app that does *not* want to use the Internet for something or another.

However, mobile Internet access inherits all of the classic problems of Internet access (e.g., "server not found") and adds new and exciting challenges, all of which can leave a developer with an app that has performance issues.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You're Using Too Much of the Slow Stuff

To paraphrase America's Founding Fathers, "all Internet connections are not created equal".

One form of inequality is speed. Different classes of connection have different theoretical upper bounds. WiMAX and other "4G" connections are theoretically faster than 3G connections, which are theoretically faster than 2G or EDGE connections. WiFi — typically 802.11g in today's devices — is theoretically ridiculously fast though it is typically limited by the ISP connection, and ISP connections can run the gamut from really fast to merely good.

However, "theoretical" bounds tend to run afoul of reality. There are plenty of places where high-speed mobile data connections are non-existent, despite what the carriers' coverage maps claim. 2G mobile data works, but is not especially speedy. This layers on top of the typical Internet congestion issues, along with typically transitory problems (e.g., trying to get connectivity while attending a technology conference keynote presentation).

Hence, what runs quickly in the lab may run much more slowly in users' hands.

If you followed the instructions in previous chapters on CPU bottlenecks, the limited bandwidth will not cause your UI to become "janky", in that it will be responsive to touches and taps. However, poor connectivity will mean that you are simply slow to respond to user requests. For example, clicking the "check for new email" menu button has no immediate effect. If you feel that you need a splash screen or progress indicator to tell the user that "we are really checking for new email, honest", then you know that your Internet access is slower than is ideal.

Obviously, some of this is unavoidable. However, the objective of the chapters in this part of the book is to give you an idea of ways to reduce your bandwidth consumption, making those delays be that much less annoying for your users.

You're Using Too Much of the Expensive Stuff

Mobile data tends to come with more strings attached than does WiFi.

In the US, it used to be that mobile data connections included unlimited usage. Now, at best, a mobile data plan has "unlimited" usage for a curious definition of the term "unlimited". More and more carriers are moving towards a hard cap — go above

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the cap, and you either cannot use more bandwidth, have your speeds curtailed, or pay significantly for additional bandwidth.

Outside of the US, the "pay significantly for bandwidth" approach is fairly typical. So-called "metered" data plans simply charge you such-and-so per MB or GB of bandwidth.

And, to top it off, roaming almost always is a metered plan. So, a US resident traveling overseas, even with a SIM and phone that supports international usage, would pay a ridiculous sum for bandwidth. Stories of phone bills in the tens of thousands of dollars abound, where people simply used their phone as they normally would when they were outside of their home network.

Hence, if you use a fair bit of bandwidth, it would be really nice if you offered users means to consume less of it when they are on mobile data compared to WiFi (which is typically unmetered). You could elect to poll your server less frequently, for example, giving the users the ability to specify separate polling periods depending on which type of connection they have.

And, of course, there are other "costs" for using bandwidth besides direct monetary costs. For example, downloading data over a slower mobile data connection may consume more power than downloading the same data over WiFi — while the WiFi radio might consume additional power, the time difference might account for more power consumption, if the CPU could be powered down for the rest of that time.

These chapters will show you how you can react to changes in connectivity and approaches for how to use that information to reduce costs for the user.

You're Using Too Much of Somebody Else's Stuff

It is easy for developers to think that they alone are using a user's device. Alas, this is infrequently the case, particularly when it comes to background Internet access.

While your application is busily downloading stuff, some other application might be busily downloading stuff. In principle, this should not be an issue, as multiple applications can access the Internet simultaneously. However, bandwidth can become an issue. If you are in the background, and the other application is in the foreground, the user might notice that bandwidth is an issue. For example, users might be unhappy if your downloads are impeding their ability to watch streaming video, or play their favorite Android-based MMORPG, or whatever.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A polite Android application will test to see whether the foreground application is heavily using the Internet and will curtail its own Internet use while that is going on. This chapter will help you learn how to make that determination and how to respond.

You're Using Too Much… And There Is None

Not only might location dictate how much bandwidth you have, but whether you have any bandwidth at all.

While some people think that the entire planet has connectivity, reality once again dictates otherwise. Major metropolitan areas have connectivity... at least, so long as the carriers have not melted down due to overuse, as AT&T tended to do during the early months of the iPhone Invasion. Outlying areas are much more hit-or-miss. Voice is sometimes a challenge, let alone data. And it only *seems* as though there is a Starbucks every 100 meters, which might actually provide blanket WiFi coverage.

Then, of course, there are planes (most do not offer in-flight WiFi at this time), international travel without an international-capable phone plan, and so on.

Some Android applications have the potential to still offer near-complete functionality despite this, with a bit of user assistance. For example, Google Maps for Android now has an offline caching feature, which will download data for a 10-mile radius from a given point, for use while the device is otherwise offline.

Here, the issue becomes less one of bandwidth (other than detecting that you have no connection) and more one of caching and storage. The space-related issues that these techniques can raise will be covered elsewhere in this book.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Focus On: TrafficStats

To be able to have more intelligent code — code that can adapt to Internet activity on the device — Android offers the `TrafficStats` class. This class really is a gateway to a block of native code that reports on traffic usage for the entire device and per-application, for both received and transmitted data. This chapter will examine how you can access `TrafficStats` and interpret its data.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

TrafficStats Basics

The `TrafficStats` class is not designed to be instantiated — you will not be invoking a constructor by calling `new TrafficStats()` or something like that. Rather, `TrafficStats` is merely a collection of static methods, mapped to native code, that provide access to point-in-time traffic values. No special permissions are needed to use any of these methods. Most of the methods were added in API Level 8 and therefore should be callable on most Android devices in use today.

Device Statistics

If you are interested in overall traffic, you will probably care most about the `getTotalRxBytes()` and `getTotalTxBytes()` on `TrafficStats`. These methods return received and transmitted traffic, respectively, measured in bytes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You also have:

1. `getTotalRxPackets()` and `getTotalTxPackets()`, if for your case measuring IP packets is a better measure than bytes
2. `getMobileRxBytes()` and `getMobileTxBytes()`, which return the traffic going over mobile data (also included in the total)
3. `getMobileRxPackets()` and `getMobileTxPackets()`, which are the packet counts for the mobile data connection

Per-Application Statistics

Technically, `TrafficStats` does not provide per-application traffic statistics. Rather, it provides per-UID traffic statistics. In most cases, the UID (user ID) of an application is unique, and therefore per-UID statistics map to per-application statistics. However, it is possible for multiple applications to share a single UID (e.g., via the `android:sharedUserId` manifest attribute) — in this case, `TrafficStats` would appear to provide traffic data for all applications sharing that UID.

There are per-UID equivalents of the first four methods listed in the previous section, replacing "Total" with "Uid". So, to find out overall traffic for an application, you could use `getUidRxBytes()` and `getUidTxBytes()`. However, these are the only two UID-specific methods that were implemented in API Level 8. Equivalents of the others (e.g., `getUidRxPackets()`) were added in API Level 12. API Level 12 also added some TCP-specific methods (e.g., `getUidTcpTxBytes()`). Note, though, that the mobile-only method are only available at the device level; there are no UID-specific versions of those methods.

Interpreting the Results

You will get one of two types of return value from these methods.

In theory, you will get the value the method calls for (e.g., number of bytes, number of packets). The documentation does not state the time period for that value, so while it is possible that it is really "number of bytes since the device was booted", we do not know that for certain. Hence, `TrafficStats` results should be used for comparison purposes, either comparing the same value over time or comparing multiple values at the same time. For example, to measure bandwidth consumption, you will need to record the `TrafficStats` values at one point in time, then again later — the difference between them represents the consumed bandwidth during that period of time.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In practice, while the "total" methods seem reliable, the per-UID methods often return `-1`. The official explanation for this is that the particular traffic metric is unavailable on that device, and this does explain some of the `-1` values that are returned. For example, a Nexus One running Android 2.3 returns `-1` for all the per-UID methods, while a Nexus S running Android 2.3 will return a positive value for *some* UIDs. It is unclear what the other `-1` values mean. Two possible meanings are:

1. There has been no traffic of that type on that UID since boot, or
2. You do not have permission to know the traffic of that type on that UID

Hence, the per-UID values are a bit "hit or miss", which you will need to take into account.

Example: TrafficMonitor

To illustrate the use of `TrafficStats` methods and analysis, let us walk through the code associated with the `Bandwidth/TrafficMonitor` sample application. This is a simple activity that records a snapshot of the current traffic levels on startup, then again whenever you tap a button. On-screen, it will display the current value, previous value, and difference ("delta") between them. In LogCat, it will dump the same information on a per-UID basis.

TrafficRecord

It would have been nice if `TrafficStats` were indeed an object that you would instantiate, that captured the traffic values at that moment in time. Alas, that is not how it was written, so we need to do that ourselves. In the `TrafficMonitor` project, this job is delegated to a `TrafficRecord` class:

```
package com.commonsware.android.tuning.traffic;

import android.net.TrafficStats;

class TrafficRecord {
  long tx=0;
  long rx=0;
  String tag=null;

  TrafficRecord() {
    tx=TrafficStats.getTotalTxBytes();
    rx=TrafficStats.getTotalRxBytes();
  }

  TrafficRecord(int uid, String tag) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    tx=TrafficStats.getUidTxBytes(uid);
    rx=TrafficStats.getUidRxBytes(uid);
    this.tag=tag;
  }
}
```

There are two separate constructors, one for the total case and one for the per-UID case. The total case just logs `getTotalRxBytes()` and `getTotalTxBytes()`, while the per-UID case uses `getUidRxBytes()` and `getUidTxBytes()`. The per-UID case also stores a "tag", which is simply a `String` identifying the UID for this record — as you will see, `TrafficMonitor` uses this for a package name.

TrafficSnapshot

An individual `TrafficRecord`, though, is insufficient to completely capture the traffic figures at a moment in time. We need a collection of `TrafficRecord` objects, one for the device ("total") and one per running UID. The work to collect all of that is handled by a `TrafficSnapshot` class:

```
package com.commonsware.android.tuning.traffic;

import java.util.HashMap;
import android.content.Context;
import android.content.pm.ApplicationInfo;

class TrafficSnapshot {
  TrafficRecord device=null;
  HashMap<Integer, TrafficRecord> apps=
    new HashMap<Integer, TrafficRecord>();

  TrafficSnapshot(Context ctxt) {
    device=new TrafficRecord();

    HashMap<Integer, String> appNames=new HashMap<Integer, String>();

    for (ApplicationInfo app :
          ctxt.getPackageManager().getInstalledApplications(0)) {
      appNames.put(app.uid, app.packageName);
    }

    for (Integer uid : appNames.keySet()) {
      apps.put(uid, new TrafficRecord(uid, appNames.get(uid)));
    }
  }
}
```

The constructor uses `PackageManager` to iterate over all installed applications and builds up a `HashMap`, mapping the UID to a `TrafficRecord` for that UID, tagged with

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the application package name (e.g., `com.commonsware.android.tuning.traffic`). It also creates one `TrafficRecord` for the device as a whole.

TrafficMonitorActivity

`TrafficMonitorActivity` is what creates and uses `TrafficSnapshot` objects. This is a fairly conventional activity with a TableLayout-based UI:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/table"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

  <Button
    android:onClick="takeSnapshot"
    android:text="Take Snapshot"/>

  <TableRow>

    <TextView
      android:layout_column="1"
      android:layout_gravity="right"
      android:text="@string/received"
      android:textSize="20sp"/>

    <TextView
      android:layout_gravity="right"
      android:text="@string/sent"
      android:textSize="20sp"/>
  </TableRow>

  <TableRow>

    <TextView
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:text="@string/latest"
      android:textSize="20sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/latest_rx"
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:textSize="20sp"/>

    <TextView
      android:id="@+id/latest_tx"
      android:gravity="right"
      android:textSize="20sp"/>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  </TableRow>

  <TableRow>

    <TextView
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:text="@string/previous"
      android:textSize="20sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/previous_rx"
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:textSize="20sp"/>

    <TextView
      android:id="@+id/previous_tx"
      android:gravity="right"
      android:textSize="20sp"/>
  </TableRow>

  <TableRow>

    <TextView
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:text="@string/delta"
      android:textSize="20sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/delta_rx"
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:textSize="20sp"/>

    <TextView
      android:id="@+id/delta_tx"
      android:gravity="right"
      android:textSize="20sp"/>
  </TableRow>

</TableLayout>
```

The activity implementation consists of three methods. There is your typical `onCreate()` implementation, where we initialize the UI, get our hands on the `TextView` widgets for output, and take the initial snapshot:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    setContentView(R.layout.main);

    latest_rx=(TextView)findViewById(R.id.latest_rx);
    latest_tx=(TextView)findViewById(R.id.latest_tx);
    previous_rx=(TextView)findViewById(R.id.previous_rx);
    previous_tx=(TextView)findViewById(R.id.previous_tx);
    delta_rx=(TextView)findViewById(R.id.delta_rx);
    delta_tx=(TextView)findViewById(R.id.delta_tx);

    takeSnapshot(null);
  }
```

The `takeSnapshot()` method creates a new `TrafficSnapshot` (held in a `latest` data member) after moving the last `TrafficSnapshot` to a `previous` data member. It then updates the `TextView` widgets for the `latest` data and, if the `previous` data member is not `null`, also for the `previous` snapshot and the difference between them. This alone is sufficient to update the UI, but we also want to log per-UID data to LogCat:

```
  public void takeSnapshot(View v) {
    previous=latest;
    latest=new TrafficSnapshot(this);

    latest_rx.setText(String.valueOf(latest.device.rx));
    latest_tx.setText(String.valueOf(latest.device.tx));

    if (previous!=null) {
      previous_rx.setText(String.valueOf(previous.device.rx));
      previous_tx.setText(String.valueOf(previous.device.tx));

      delta_rx.setText(String.valueOf(latest.device.rx-previous.device.rx));
      delta_tx.setText(String.valueOf(latest.device.tx-previous.device.tx));
    }

    ArrayList<String> log=new ArrayList<String>();
    HashSet<Integer> intersection=new HashSet<Integer>(latest.apps.keySet());

    if (previous!=null) {
      intersection.retainAll(previous.apps.keySet());
    }

    for (Integer uid : intersection) {
      TrafficRecord latest_rec=latest.apps.get(uid);
      TrafficRecord previous_rec=
            (previous==null ? null : previous.apps.get(uid));

      emitLog(latest_rec.tag, latest_rec, previous_rec, log);
    }

    Collections.sort(log);

    for (String row : log) {
      Log.d("TrafficMonitor", row);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    }
  }
```

One possible problem with the snapshot system is that the process list may change between snapshots. One simple way to address this is to only log to LogCat data where the application's UID exists in both the `previous` and `latest` snapshots. Hence, `takeSnapshot()` uses a `HashSet` and `retainAll()` to determine which UIDs exist in both snapshots. For each of those, we call an `emitLog()` method to record the data to an `ArrayList`, which is then sorted and dumped to LogCat.

The `emitLog()` method builds up a line with the package name and bandwidth consumption information, assuming that there is bandwidth to report (i.e., we have a value other than `-1`):

```
  private void emitLog(CharSequence name, TrafficRecord latest_rec,
                       TrafficRecord previous_rec,
                       ArrayList<String> rows) {
    if (latest_rec.rx>-1 || latest_rec.tx>-1) {
      StringBuilder buf=new StringBuilder(name);

      buf.append("=");
      buf.append(String.valueOf(latest_rec.rx));
      buf.append(" received");

      if (previous_rec!=null) {
        buf.append(" (delta=");
        buf.append(String.valueOf(latest_rec.rx-previous_rec.rx));
        buf.append(")");
      }

      buf.append(", ");
      buf.append(String.valueOf(latest_rec.tx));
      buf.append(" sent");

      if (previous_rec!=null) {
        buf.append(" (delta=");
        buf.append(String.valueOf(latest_rec.tx-previous_rec.tx));
        buf.append(")");
      }

      rows.add(buf.toString());
    }
  }
```

Since the lines created by `emitLog()` start with the package name, and since we are sorting those before dumping them to LogCat, they appear in LogCat in sorted order by package name.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Using TrafficMonitor

Running the activity gives you the initial received and sent counts (in bytes):

Figure 772: The TrafficMonitor sample application, as initially launched

Tapping Take Snapshot grabs a second snapshot and compares the two:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 773: The TrafficMonitor sample application, after Take Snapshot was clicked

Also, LogCat will show how much was used by various apps:

```
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283):
com.amblingbooks.bookplayerpro=880 received (delta=0), 3200 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.android.browser=19045241
received (delta=0), 2375847 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283):
com.android.providers.downloads=27884469 received (delta=0), 9126 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283):
com.android.providers.telephony=2328 received (delta=0), 4912 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.android.vending=3271839
received (delta=0), 260626 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.coair.mobile.android=887425
received (delta=0), 81366 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.commonsware.android.browser1=262553 received (delta=0), 7286 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.dropbox.android=6189833
received (delta=0), 4298 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.evernote=3471398 received
(delta=0), 742178 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.apps.genie.geniewidget=358816 received (delta=0), 17775 sent
(delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.apps.googlevoice=103255 received (delta=0), 35559 sent
(delta=0)
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.apps.maps=28440829 received (delta=0), 1230867 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.backup=51320
received (delta=0), 49041 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.gm=10915084
received (delta=0), 14428803 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.googlequicksearchbox=37817 received (delta=0), 12554 sent
(delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.syncadapters.contacts=1955990 received (delta=0), 714893 sent
(delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.voicesearch=67948 received (delta=0), 121908 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.youtube=3128
received (delta=0), 2792 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.howcast.android.app=2250407
received (delta=0), 26727 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.rememberthemilk.MobileRTM=6836605 received (delta=0), 2902904 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.tripit=109499 received
(delta=0), 50060 sent (delta=0)
```

Other Ways to Employ TrafficStats

Of course, there are more ways you could use `TrafficStats` than simply having an activity to report them on a button click. `TrafficMonitor` is merely a demonstration of using the class and providing a lightweight way to get value out of that data. Depending upon your application's operations, though, you may wish to consider using `TrafficStats` in other ways, in your production code or in your test suites.

In Production

If your app is a bandwidth monitor, the need to use `TrafficStats` is obvious. However, even if your app does something else, you may wish to use `TrafficStats` to understand what is going on in terms of Internet access within your app or on the device as a whole.

For example, you might want to consider bandwidth consumption to be a metric worthy of including in the rest of the "analytics" you generate from your app. If you are using services like Flurry to monitor which activities get used and so on, you might consider also logging the amount of bandwidth your application consumes. This not only gives you much more "real world" data than you will be able to collect on your own, but it may give you ideas of how users are using your application beyond what the rest of your metrics are reporting.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Another possibility would be to include your app's bandwidth consumption in error logs reported via libraries like ACRA. Just as device particulars can help identify certain bug report patterns, perhaps certain crashes of your app only occur when users are using a lot of bandwidth in your app, or using a lot of bandwidth elsewhere and perhaps choking your own app's Internet access.

The chapter on bandwidth mitigation strategies will also cover a number of uses of `TrafficStats` for real-time adjustment of your application logic.

During Testing

You might consider adding `TrafficStats`-based bandwidth logging for your application in your test suites. While individual tests may or may not give you useful data, you may be able to draw trendlines over time to see if you are consuming more or less bandwidth than you used to. Take care to factor in that you may have changed the tests, in addition to changing the code that is being tested.

From a JUnit-based unit test suite, measuring bandwidth consumption is not especially hard. You can bake it into the `setUp()` and `tearDown()` methods of your test cases, either via inheritance or composition, and log the output to a file or LogCat.

From an external test engine, like monkeyrunner or NativeDriver, recording bandwidth usage is more tricky, because your test code is not running on the device or emulator. You may have to include a `BroadcastReceiver` in your production code that will log bandwidth usage and trigger that code via the `am broadcast` shell command.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Measuring Bandwidth Consumption

The first step towards addressing bandwidth concerns is to get a better picture of how much bandwidth you are actually consuming, when, and under what conditions. Only then will you be able to determine where your efforts need to be applied and whether those efforts are actually giving you positive results. This chapter will examine a handful of ways you can determine how much bandwidth you are really using in your application.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

On-Device Measurement

Many times, you are best served by measuring your bandwidth consumption right on the device itself:

1. This is your only option for gathering bandwidth metrics from copies of your app in end users' hands, unless they invite you to their home or office and have you sniff on their personal network, which seems unlikely
2. This is your only option for gathering bandwidth metrics when you are using mobile data plans (e.g., 3G) instead of WiFi, since you probably do not control the wireless telecommunications infrastructure in your area
3. This is your simplest option for tying bandwidth metrics to events within your app or occurring on the device

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

4. This is your only option for using bandwidth metrics to adjust your application behavior in real time, in addition to using the metrics to learn how best to adjust your code in future updates to the app

Hence, in addition to perhaps other off-device techniques, you really should consider one of the on-device approaches outlined in the following sections.

Yourself, via TrafficStats

The preceding chapter outlined how to use the `TrafficStats` class to collect metrics on the bandwidth consumed by applications (including yours) and for the device as a whole. This gives you the most flexibility, because you can write your own code to collect whatever portion of this data you need. It can address all of the bullets shown above, for example.

It is not perfect, though:

1. It requires you to write your own code, adding yet more work to your plate
2. Per-UID traffic data may or may not be available, depending upon the device

Existing Android Applications

If you do not want to write code to use `TrafficStats`, there are various applications on the Play Store that can report that data to you, much along the lines of how `TrafficMonitor` does. Here are some notes about a few free ones tested by the author:

1. Network Traffic Detail (v. 1.3) works, but does not consider that bandwidth is only reported per UID, not per application. As a result, it reports the same traffic multiple times, one for each application sharing a UID.
2. Traffic Monitor (v. 2.4.2) advertises itself as an application, but does not put an icon in the launcher for it, forcing you to install an app widget instead in order to get to the actual application. While it reports device-level bandwidth, and it has a task manager, the task list does not report bandwidth for those tasks.
3. Bandwidth Monitor (v. 1.0.6) works and is perhaps incrementally easier to use than the other alternatives, though its touted bar chart of bandwidth consumption lacks any indicator of the value of the Y axis.

There are certainly others on the Market today and more will show up over time. For your own use, these sorts of apps may be very helpful. However, since you control

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

nothing over what is collected and how (and, in the case of some, even when it is collected), it may be difficult for you to get a solid grasp on where your code is consuming bandwidth this way.

There are also various apps that provide more in the way of packet-sniffing capability. However, these require you to root your phone and run the app with root privileges.

Off-Device Measurement

The biggest limitation of `TrafficStats` is that it only gives you gross metrics: numbers of bytes, packets, and so on. Sometimes, that is not enough to help you understand why those bytes, packets, and so on are actually being sent or received. Sometimes, it would be nice to understand the traffic in more detail, from the ports and IP addresses to perhaps the actual data being transmitted. For obvious security reasons, this is not something an ordinary Android SDK application can do. However, there are techniques for accomplishing this, mostly for use over WiFi in your own home or office network. Some of these are outlined in the following sections.

Wireshark

Wireshark, formerly known as Ethereal, is perhaps the world's leading open source network traffic analyzer and packet inspector. Using it, you can learn in great detail what is going on with your local network. And, Android provides additional options for you to leverage Wireshark to make sense of application behavior. Wireshark is available for Linux, OS X, and Windows.

There is a lightly-documented `-tcpdump` switch available on the Android emulator. If you launch the emulator from the command line with that switch (plus `-avd` to identify the AVD file you want to use), all network access is dumped to your specified log file. You can then load that data into Wireshark for analysis, via File|Open from the main menu.

For example, here is a screenshot of Wireshark examining data from such an emulator dump file, in which the emulator was used to conduct a Google search:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

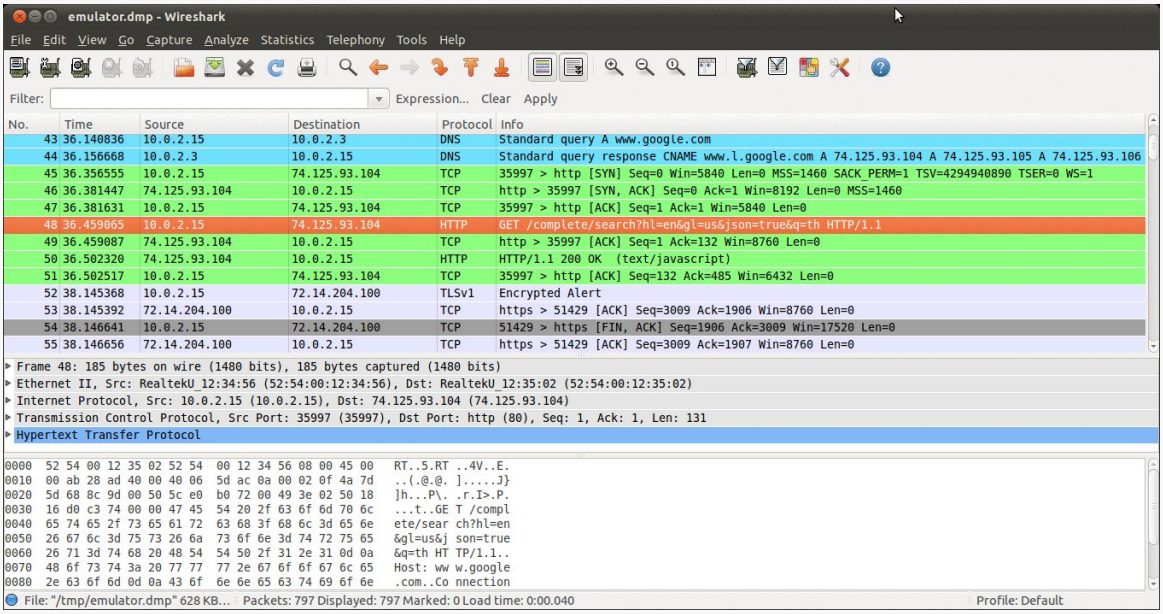


Figure 774: Wireshark examining captured emulator packets

This screenshot shows an HTTP request in the highlighted line in the list, with the hex and ASCII contents of the request shown in the bottom pane.

In terms of using Wireshark to monitor traffic from actual hardware, that is indubitably possible. However, WiFi packet collection is a tricky process with Wireshark, being very dependent upon operating system and possibly even the WiFi adapter chipset. You also get much lower-level information, making it a bit more challenging to figure out what is going on. Attempting to cover all of this is well beyond the scope of this book and the author's Wireshark expertise.

Networking Hardware

Sophisticated firewalls sometimes have packet tracing/sniffing capability. In this case, "sophisticated" does not necessarily mean "expensive", as open source router/firewall distributions, like OpenWrt, can be used for this sort of work. In this case, the router captures the packets and, in many cases, routes them to Wireshark for analysis. Some might offer on-board analysis (e.g., Web interface to packet capture logs).

This is particularly useful on a Windows wireless network. Wireshark has limits, imposed by Windows, that cause some problems when trying to capture WiFi

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

packets. By offloading the packet capture to networking hardware, those limits can be bypassed.

Tactical Measurement in DDMS

`TrafficStats` is great for measuring gross bandwidth consumption over some period of time. However, it requires coding, logging, and your own analysis mechanism.

Another approach is to use the new Network Statistics view available as part of DDMS. This view will report, in real time, what your receiving and transmitting bandwidth usage is, in the form of a line chart. This tool was added to the r17 edition of the Android tools.

To use this view, you will need a device running Android 4.0.3 or higher. It does not work with the emulator or older devices, unfortunately.

If you have such a device though, when you run your app on it, you can:

- Open the Network Statistics view in Eclipse (or the equivalent in the standalone `monitor` tool)
- Run your app and get it ready for testing
- Click on your debuggable process in the Devices view
- Choose a refresh speed for the line chart (100ms, 250ms, or 500ms) in the Network Statistics view
- Click the Start button adjacent to the speed drop-down
- Do something in your app to trigger network I/O
- Click the Stop button to freeze the updates to the line chart

What you will get, out of the box, is something like this:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

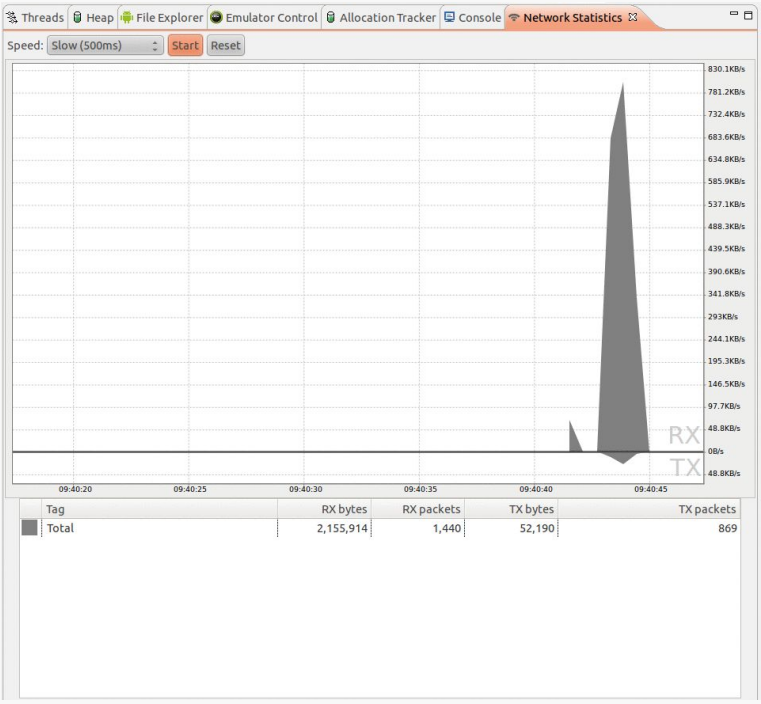


Figure 775: Network Statistics View

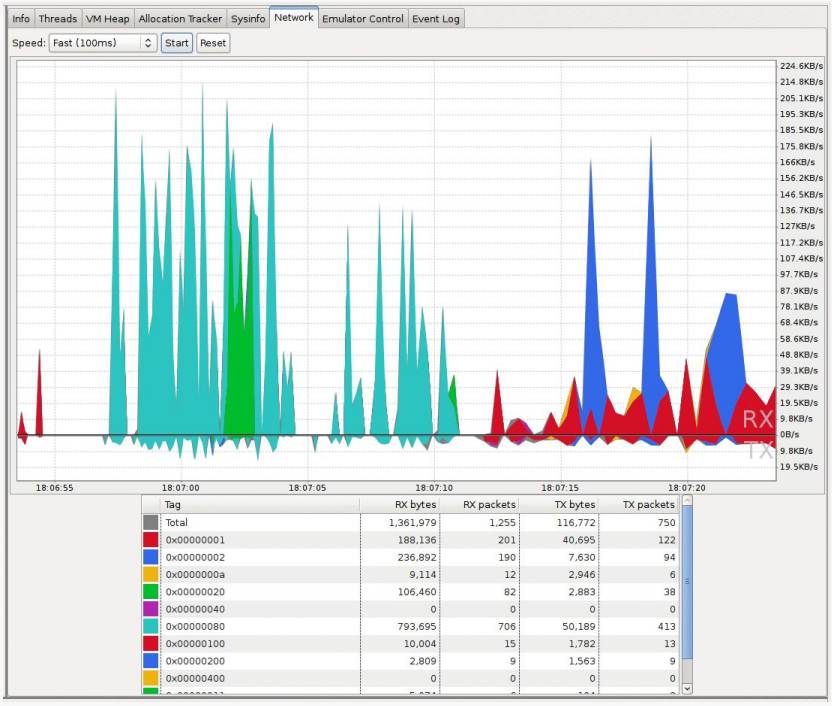
This particular output came from a run of the `DownloadItYourself` demo app from earlier in the book. The table at the bottom shows the total amount of bandwidth consumed during the test run, and the line chart at the top helps to illustrate when we consumed that bandwidth. Received bandwidth appears above the baseline; transmitted bandwidth appears below the baseline.

For fairly simple cases, this is all you will need. If, however, you have lots of things going on, you might want to track individual bits of network I/O. Android supports a tagging concept that will help with this, allowing you to associate a tag with the current thread.

In most cases, if you are using higher-level libraries like `HttpUrlConnection` or `HttpClient`, you would use `setThreadStatsTag()`, a static method added to `TrafficStats` in API Level 14. You supply an integer "tag", which will be associated with network I/O performed on that tag from those libraries. If you are working with raw sockets, you will also need to use `tagSocket()` and `untagSocket()` to associate the work for that socket with the tag for the current thread.

This will then give you a much more detailed set of output, showing not only total network I/O but per-tag network I/O:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition



Tag	RX bytes	RX packets	TX bytes	TX packets
Total	1,361,979	1,255	116,772	750
0x00000001	188,136	201	40,695	122
0x00000002	236,892	190	7,630	94
0x0000000a	9,114	12	2,946	6
0x00000020	106,460	82	2,883	38
0x00000040	0	0	0	0
0x00000080	793,695	706	50,189	413
0x00000100	10,004	15	1,782	13
0x00000200	2,809	9	1,563	9
0x00000400	0	0	0	0

Figure 776: Network Statistics Detailed View (image courtesy of Android Open Source Project)

Clicking the Reset button, to the right of the Start button, clears the graph and table, to give you fresh results for your next test.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Being Smarter About Bandwidth

Given that you are collecting metrics about bandwidth consumption, you can now start to determine ways to reduce that consumption. You may be able to permanently reduce that consumption (at least on a per-operation basis). You may be able to shunt that consumption to times or networks that the user prefers. This chapter reviews a variety of means of accomplishing these ends.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate, particularly the chapter on Internet access.

Bandwidth Savings

The best way to reduce bandwidth consumption is to consume less bandwidth.

(in other breaking news, water is wet)

In recent years, developers have been able to be relatively profligate in their use of bandwidth, pretty much assuming everyone has an unlimited high-speed Internet connection to their desktop or notebook and the desktop or Web apps in use on them. However, those of us who lived through the early days of the Internet remember far too well the challenges that dial-up modem accounts would present to users (and perhaps ourselves). Even today, as Web apps try to "scale to the Moon and back", bandwidth savings becomes important not so much for the end user, but for the Web app host, so its own bandwidth is not swamped as its user base grows.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Fortunately, widespread development problems tend to bring rise to a variety of solutions — a variant on the "many eyes make bugs shallow" collaborative development phenomenon. Hence, there are any number of tried-and-true techniques for reducing bandwidth consumption that have had use in Web apps and elsewhere. Many of these are valid for native Android apps as well, and a few of them are profiled in the following sections.

Classic HTTP Solutions

Trying to get lots of data to fit on a narrow pipe — whether that pipe is on the user's end or the provider's end — has long been a struggle in Web development. Fortunately, there are a number of ways you can leverage HTTP intelligently to reduce your bandwidth consumption.

GZip Encoding

By default, HTTP requests and response are uncompressed. However, you can enable GZip encoding and thereby request that the server compress its response, which is then decompressed on the client. This trades off CPU for bandwidth savings and therefore needs to be done judiciously.

Enabling GZip compression is a two-step process:

- Adding the `Accept-Encoding: gzip` header to the HTTP request
- Determine if the response was compressed and, if so, decompressing it

Bear in mind that the Web server may or may not honor your GZip request, for whatever reason (e.g., response is too small to make it worthwhile).

For example, using the HttpClient library in Android, you could add the header on the request:

```
HttpGet get=new HttpGet(url);

get.addHeader("Accept-Encoding", "gzip");

// rest of configuration here, if any

// execute the request given an HttpClient object named client
HttpResponse response=client.execute(get);
```

Then, you can check the response and get a valid `InputStream` for either the compressed or the not-compressed cases:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
// assumes HttpResponse response as in above code snippet

InputStream stream=response.getEntity().getContent();
Header enc=response.getFirstHeader("Content-Encoding");

if (enc!=null && enc.getValue().equalsIgnoreCase("gzip")) {
    stream=new GZIPInputStream(stream);
}

// at this point, stream will work for either encoding
```

Equivalents exist for using `HttpUrlConnection`, if you prefer to use that HTTP API in Android.

If-Modified-Since / If-None-Match

Of course, avoiding a download offers near–100% compression. If you are caching data, you can take advantage of HTTP headers to try to skip downloads that are the same content as what you already have, specifically `If-Modified-Since` and `If-None-Match`.

An HTTP response can contain either a `Last-Modified` header or an `ETag` header. The former will contain a timestamp and the latter will contain some opaque value. You can store this information with the cached copy of the data (e.g., in a database table). Later on, when you want to ensure you have the latest version of that file, your HTTP GET request can include an `If-Modified-Since` header (with the cached Last-Modified value) or an `If-None-Match` header (with the cached `ETag` value). In either case, the server should return either a `304` response, indicating that your cached copy is up to date, or a `200` response with the updated data. As a result, you avoid the download entirely (other than HTTP headers) when you do not need the updated data.

For example, using HttpClient, you can check for the existence of an `ETag` header in an HTTP response:

```
HttpGet get=new HttpGet(url);

// execute the request given an HttpClient object named client
HttpResponse response=client.execute(get);
Header etag=response.getFirstHeader("ETag");

if (etag!=null) {
  // cache this
}

// process the download
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

On subsequent requests, you can add the `If-None-Match` header and handle both cases:

```
HttpGet get=new HttpGet(url);

get.addHeader("If-None-Match", etag);

// execute the request given an HttpClient object named client
HttpResponse response=client.execute(get);
int sc=response.getStatusLine().getStatusCode();

if (sc!=HttpStatus.SC_NOT_MODIFIED) {
  // cache invalid, so process the download and, perhaps, grab fresh
  // ETag
}
```

Using `Last-Modified` and `If-Modified-Since` is mostly a matter of switching headers. And, once again, there are equivalent ways to use these headers with `HttpUrlConnection`.

Binary Payloads

While XML and JSON are relatively easy for humans to read, that very characteristic means they tend to be bloated in terms of bandwidth consumption. There are a variety of tools, such as Google's Protocol Buffers and Apache's Thrift, that allow you to create and parse binary data structures in a cross-platform fashion. These might allow you to transfer the same data that you would in XML or JSON in less space. As a side benefit, parsing the binary responses is likely to be faster than parsing XML or JSON. Both of these tools involve the creation of an IDL-type file to describe the data structure, then offer code generators to create Java classes (or equivalents for other languages) that can read and write such structures, converting them into platform-neutral on-the-wire byte arrays as needed.

Minification

If you are loading JavaScript or CSS into a `WebView`, you should consider standard tricks for compressing those scripts, collectively referred to as "minification". These techniques eliminate all unnecessary whitespace and such from the files, rename variables to be short, and otherwise create a syntactically-identical script that takes up a fraction of the space. There are services like box.js that can even aggregate several scripts into one file and minify them, to further reduce HTTP overhead.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Push versus Poll

Another way to consume less bandwidth is to only make the requests when it is needed. For example, if you are writing an email client, the way to use the least bandwidth is to download new messages only when they exist, rather than frequently polling for messages.

Off the cuff, this may seem counter-intuitive. After all, how can we know whether or not there are any messages if we are not polling for them?

The answer is to use a low-bandwidth push mechanism. The quintessential example of this is GCM, the Google Cloud Messaging system, available for Android 2.2 and newer. This service from Google allows your application to subscribe to push notifications sent out by your server. Those notifications are delivered asynchronously to the device by way of Google's own servers, using a long-lived socket connection. All you do is register a `BroadcastReceiver` to receive the notifications and do something with them.

For example, Remember the Milk — a task management Web site and set of mobile apps — uses GCM to alert the device of task changes you make through the Web site. Rather than the Remember the Milk app having to constantly poll to see if tasks were added, changed, or deleted, the app simply waits for GCM events.

You could create your own push mechanism, perhaps using a WebSocket or Comet-style long-poll technique. The downside is that you will need a service in memory all of the time to manage the socket and thread that monitors it. If you only need this while your service is in memory for other reasons, that is fine. However, keeping a service in memory 24x7 has its own set of issues, not the least of which is that users will tend to smack it down using a "task killer" or the Manage Services screen in the Settings app.

Thumbnails and Tiles

A general rule of thumb is: don't download it until you really need it.

Sometimes, you do not know if you really need a particular item until something happens in the UI. Take a `ListView` displaying thumbnails of album covers for a music app. Assuming the album covers are not stored locally, you will need to download them for display. However, which covers you need varies based upon scrolling. Downloading a high-resolution album cover that might get tossed in a

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

matter of milliseconds (after an expensive rescale to fit a thumbnail-sized space) is a waste of bandwidth.

In this case, either the album covers are something you control on the server side, or they are not. If they are, you can have the server prepare thumbnails of the covers, stored at a spot that the app can know about (e.g., `.../cover.jpg` it is `.../thumbnail.jpg`). The app can then download thumbnails on the fly and only grab the full-resolution cover if needed (e.g., user clicks on the album to bring up a detail screen). If you do not control the album covers, this option might still be available to you if you can run your own server for the purposes of generating such thumbnails.

You can see a similar effect with the map tiles in Google Maps. When zooming out, the existing map tiles are scaled down, with placeholders (the gridlines) for the remaining spots, until the tiles for those spots are downloaded. When zooming in, the existing map tiles are scaled up with a slight blurring effect, to give the user some immediate feedback while the full set of more-detailed tiles is downloaded. And, if the user pans, you once again get placeholders while the tiles for the newly uncovered areas are downloaded. In this fashion, Google Maps is able to minimize bandwidth consumption by giving users partial results immediately and back-filling in the final results only when needed. This same sort of approach may be useful with your own imagery.

Collaborative Bandwidth

For some common services, perhaps sharing is the best option to reduce bandwidth usage.

For example, consider Twitter. It is entirely possible that a user might have multiple applications all polling and downloading the user's timeline:

1. A built-in Twitter app that the user does not like, but cannot uninstall
2. A regular Twitter app that the user employs for normal stuff
3. A separate Twitter app widget, because the other Twitter apps on the device either lack an app widget or the user does not like it
4. Yet another application that uses Twitter as one of several data sources (e.g., monitoring for references to certain keywords, such as a company name, across multiple social networks)

In an ideal world, all of these apps would use one common engine that handles collecting the tweets and making them available — securely — to the other

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

applications. This would dramatically cut bandwidth by eliminating redundant polling.

If your data source is used by other applications, consider reaching out to those developers and creating a common engine, perhaps using a `ContentProvider` for data sharing, an `IntentService` or sync provider for collecting the data, plus common activities for preferences. Distribute the code to all of the development teams as an Android library project. Ship these components disabled in your manifest, enabling them if you cannot find another implementation on the device, indicating that you are the only one of this "application family" installed. If you do find another implementation, use that one instead of your own. There are certainly issues to be dealt with here (e.g., what if the user uninstalls the app that the others are depending upon), but it is worth considering for shared development costs as well as shared bandwidth.

Bandwidth Shaping

Sometimes, you have no ability to reduce the bandwidth itself. Perhaps you do not control both ends of the communications pipeline. Perhaps the data you are trying to exchange is already compressed (e.g., downloading an MP4 video). Perhaps some of the techniques in the preceding section were unavailable to you (e.g., cannot route data through third-party servers like Google's for GCM).

There still may be ways for you to help your users, by shaping your bandwidth use. Rather than just blindly doing whatever you want whenever you want, you learn what the *user* wants and what *other applications* want and tailor your bandwidth use on the fly to match those needs. The following sections outline some ways of achieving this.

Driven by Preferences

If you are consuming enough bandwidth that this chapter is relevant to you, you probably are consuming enough bandwidth that you should be asking the user how best to consume that bandwidth. After all, they are the one paying the price — in time as well as money – for that consumption.

The following sections present some possible strategies for preference-based bandwidth shaping.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Budgets

One strategy is for the user to give you a budget (e.g., 20MB/day) and for you to stick within that budget.

Collecting the budget is fairly easy — just use `SharedPreferences`. Either use a `ListPreference` with likely budget value or an `EditTextPreference` and a bit of validation for a free-form budget amount.

Next, you will need to have some idea how much bandwidth any given network operation will consume. For some things, this might be an estimate based on your experiments as a developer, or perhaps it is based on historical averages for this user and type of operation. For example, a "podcatcher" (feed reader designed to download podcast episodes) should have some idea how big a given RSS or Atom feed download should be. In some cases, it might be worthwhile to get a better estimate — for example, the podcatcher might use an HTTP `HEAD` request to determine the size of the MP3 or OGG file before deciding whether to download it.

Then, you need to be keeping track of your budget. This could be a simple flat file with the initial `TrafficStats` bandwidth values for your process. Re-initialize that file on the first network operation of the day (or whatever period you chose for your budget). Before doing another network operation, compare the current `TrafficStats` values with the initial ones and see how close you are to the budget. If the new network operation will exceed the budget, skip the operation, perhaps putting it in a work queue to perform in the next budget. You might even hold a reserve for certain types of operations. For example, the podcatcher might ensure there is at least 10% of the budget available for downloading the feeds, even if it means putting a podcast on the queue for download tomorrow. That way, you can present to the user the latest podcast information, with icons indicating which are downloaded and which are queued for download — the user might be able to then request to override the budget and download something on demand.

For devices that lack per-UID `TrafficStats` support, you will have to "fake it" a bit. Use your own calculations of how much bandwidth each operation consumes and track that information, even if you wind up missing out on some bytes here or there.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Connectivity

If the user might not care how much bandwidth you consume, so long as it is unmetered bandwidth, you might include a `CheckBoxPreference` to indicate if large network operations should be limited to WiFi and avoid mobile data.

You could then use `ConnectivityManager` and `getActiveNetworkInfo()` to see what connection you have before performing a network operation. If it is a background operation (e.g., the podcatcher checking for new podcasts every hour), if the network is not the desired one, you can skip the operation or put it on a work queue for re-trying later. If it is a foreground operation (e.g., the user clicked a "refresh" menu choice), you could pop up a confirmation `AlertDialog` to warn the user that they are on mobile data — perhaps this time they are interested in doing the operation anyway.

Another approach for handling the background operations is to register a `BroadcastReceiver` for the `CONNECTIVITY_ACTION` broadcast (defined on `ConnectivityManager`). If the connectivity switches to mobile data, cancel your outstanding `AlarmManager` alarms; if connectivity switches to WiFi, re-enable those alarms.

Of course, you should also consider monitoring the background data setting — the global Settings checkbox indicating whether background network operations are allowed. On `ConnectivityManager`, `getBackgroundDataSetting()` tells you the state of this checkbox, and `ACTION_BACKGROUND_DATA_SETTING_CHANGED` allows you to set up a `BroadcastReceiver` to watch for changes in its state.

Windows

If your user is less concerned about the bandwidth or the network, but does care about the time of day (e.g., does not want your application consuming significant bandwidth when they might be getting a VOIP call), you could offer preferences for that as well. Cook up a `TimePreference` and use that to collect start and stop times for the high-bandwidth window. Then, set up alarms with `AlarmManager` for those points in time. The alarm for the start time of the window sets up a third alarm with your regular polling interval. The alarm for the stop time of the window cancels the polling interval alarm.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Driven by Other Usage

If your network I/O is part of a foreground application, one presumes that you are the most important thing in the user's life right now. Or, at least, the most important thing on the user's phone right now. Hence, what other applications might want to do with the Internet connection is not a major concern.

If, however, your network I/O is part of a background operation, it might be nice to try to avoid doing things that might upset the user. If the user is watching streaming video or is on a VOIP call or otherwise is aware of bandwidth changes, the bandwidth you use might impact the user in ways that the user will not appreciate very much. This is unlikely to be a big problem for small operations (e.g., downloading a 1KB JSON file), but larger operations (e.g., downloading a 5MB podcast) might be more noticeable.

You can use `TrafficStats` to help here. Before doing the actual network I/O, grab the current traffic data, wait a couple of seconds, and compare the latest to the previous values. If little to no bandwidth was consumed during that period, assume it is safe and go ahead and do your work. If, however, a bunch of bandwidth was consumed, you might want to consider:

1. Skipping this polling cycle and trying again later, or
2. Adding a one-off alarm using set() on `AlarmManager` to give you control again in a minute, with the current traffic data packaged as an extra on the `Intent`, so you can make a decision after a bigger sample size of bandwidth consumption, or
3. Adding an entry in a persistent work queue, so you know later on to try again if bandwidth contention has improved

You could try to get more sophisticated, by using `ActivityManager` and the per-UID values from `TrafficStats` to see if it is a foreground application that is the one consuming the bandwidth. It is unclear how reliable this will be, both in determining who is consuming the bandwidth (again, per-UID traffic is not available on many devices) and in avoid user angst. It may be simpler just to assume the worst and side-step your I/O until the other apps have quieted down.

Avoiding Metered Connections

Android 4.1 (a.k.a., Jelly Bean) added `isActiveNetworkMetered()` as a method on `ConnectivityManager`. In principle, this will return `true` if Android thinks that the

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

current data connection may involve bandwidth charges. You can examine this value and steer your bandwidth consumption accordingly.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Issues with Application Heap

RAM. Developers nowadays are used to having lots of it, and a virtual machine capable of using as much of it as exists (and more, given swap files and page files).

"Graybeards" — like the author of this book — distinctly remember a time when we had 16KB of RAM and were happy for it. Such graybeards would also appreciate it if you would get off their respective lawns.

Android comes somewhere in the middle. We have orders of magnitude more RAM than, say, the TRS–80 Model III. We do not have as much RAM as does the modern notebook, let alone a Web server. As such, it is easy to run out of RAM if you do not take sufficient care.

There are two facets of memory issues with Android:

- What are the problems we encounter inside our own app, in terms of our application heap?
- What problems can we encounter with system RAM overall, and how can we resolve them?

This part of the book examines memory-related issues, with this chapter focusing on the application heap. Another chapter will deal with system RAM issues. These are not to be confused with any memory-related issues inherent to graybeards.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate, particularly the chapter on Android's process model.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You Are in a Heap of Trouble

When we think of "memory" and Java-style programming, the primary form of memory is the heap. The heap holds all of our Java objects – from an `Activity` to a widget to a `String`.

Traditional Java applications have an initial heap size determined by the virtual machine, possibly configured via command-line options when the program was run. Traditional Java applications can also request additional memory from the OS, up to some maximum, also configurable.

Android applications have the same basic structure, with very limited configurability and much lower maximums than you might expect.

Older Android devices, particularly those with HVGA screens like the T-Mobile G1, tend to have a maximum of 16MB of heap space. Newer Android phones with higher-resolution screens might have 24MB (Motorola DROID) or 32MB (Nexus One) of heap space. Tablets might have 48MB of heap space.

This heap limit can be problematic. For example, each widget or layout manager instance takes around 1KB of heap space. This is why `AdapterView` provides the hooks for view recycling — we cannot have a `ListView` with literally thousands of row views without potentially running out of heap.

API Level 11+ supports applications requesting a "large heap". This is for applications that specifically need tons of RAM, such as an image editor to be used on a tablet. This is not for applications that run out of heap due to leaks or sloppy programming. Bear in mind that users will feel effects from large-heap applications, in that their other applications will be kicked out of memory more quickly, possibly irritating them. Also, garbage collection on large-heap applications runs more slowly, consuming more CPU time. To enable the large heap, add `android:largeHeap="true"` to the `<application>` element of your manifest.

Determining Your Heap Size At Runtime

To get a sense for how much heap you will be able to potentially grow to, you can call `getMemoryClass()` on an `ActivityManager`. This will return your per-process heap limit in megabytes.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you requested `android:largeHeap="true"` in the manifest, use `getLargeMemoryClass()` on `ActivityManager` to learn how large your "large heap" actually is. Note that it is entirely possible that the "large heap" is not all that large, or potentially is no bigger than the standard heap, depending upon how much RAM is physically present on the device.

Fragments of Memory

The Dalvik garbage collector is a non-compacting implementation, which makes `OutOfMemoryError` messages somewhat more likely than you would find on traditional Java environments.

Here, "non-compacting" means that Dalvik does not try to move objects around in physical memory to "compact" the use of physical memory, leaving a large contiguous block of free physical memory for future allocations.

For example, suppose that we allocate three 1K byte arrays, named A, B, and C. As it turns out, they were allocated using adjacent portions of physical memory, so that the last byte of A immediately precedes the first byte of B, and so on. Hence, we consumed 3K of available heap space to create these three 1K blocks.

If we release all references to A and B, they can be garbage-collected. Dalvik, like Java, will see that A and B are adjacent and will free up their physical memory, such that the memory is available as one contiguous 2K block for future allocations.

If, however, released all references to A and C instead of A and B, Dalvik would be unable to make their blocks be contiguous, and so our heap would have two free 1K blocks, in addition to whatever other free memory that the heap already had.

Hence, allocating memory not only ties up that memory while it is in use, but it may *fragment* the memory even when it is released, such that our formerly pristine heap is now comprised of lots of little free blocks of space, separated from other such blocks by in-use objects. When we try to make a large allocation, such as setting up a byte array for a large image, it may be that while we have enough *total* heap available for the request, there is no single block that would meet our request, and so we get an `OutOfMemoryError`.

One technique to help address this is to pre-allocate any large buffers that you know you need, up front when your process starts up, such as via a custom `Application` subclass. Then, use an "object pool" approach to obtain, use, and reuse these pre-

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

allocated buffers, rather than having them be garbage-collected and have to be re-allocated later.

Getting a Trim

It would be nice if we knew when a good time would be to cut back on our heap usage. For example, if we are caching a lot of data in our process, to save on future disk I/O, we could free up those caches at some point to help minimize our heap usage.

Fortunately, Android has some hooks for doing just that.

onTrimMemory() Callbacks

Starting in API Level 14, your activities, services, content providers, and custom `Application` classes all offer an `onTrimMemory()` method that you can override. This will be called from time to time to let you know about changes in the state of your app that might indicate it is time to free up some caches or otherwise cut back on memory consumption.

`onTrimMemory()` is passed a "level", indicating how serious the memory crunch is. At the present time, there are seven such levels, but others may be added in future versions of Android. However, these levels are in priority order, and the documentation indicates that Google will ensure that future levels are slotted into the order as appropriate. Hence, you can watch for levels of a certain severity *or higher* and take appropriate action at those points in time.

The seven levels are all defined as constants on the `ComponentCallbacks2` interface that defines `onTrimMemory()`. Four were defined in API Level 14, while the remaining three were defined in API Level 16. They are (in order of increasing severity):

- `TRIM_MEMORY_RUNNING_MODERATE` (added in API Level 16)
- `TRIM_MEMORY_RUNNING_LOW` (added in API Level 16)
- `TRIM_MEMORY_RUNNING_CRITICAL` (added in API Level 16)
- `TRIM_MEMORY_UI_HIDDEN` (added in API Level 14)
- `TRIM_MEMORY_BACKGROUND` (added in API Level 14)
- `TRIM_MEMORY_MODERATE` (added in API Level 14)
- `TRIM_MEMORY_COMPLETE` (added in API Level 14)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In particular, `TRIM_MEMORY_BACKGROUND` (or higher) indicates that your process is now on the list of processes to terminate to free up memory, and so the more memory you can free up, the less likely it is that your process will be terminated. Also, at `TRIM_MEMORY_UI_HIDDEN` or higher, your UI is no longer visible to the user, and so this is a fine time to free up UI-related memory that is safe to release, such as perhaps widget hierarchies that you would be rebuilding in `onResume()` later on anyway.

Note that while the focus tends to be on activities implementing `onTrimMemory()` to clean up UI-related resources, you are welcome to implement `onTrimMemory()` in services, content providers, and any custom `Application` subclass, so that you can free up memory that those may be managing as caches.

In the chapter on system RAM, we will get into why freeing up memory may help keep your process around, as we discuss the relationship between your application heap and available system RAM.

Warning: Contains Graphic Images

However, the most likely culprit for `OutOfMemoryError` messages are bitmaps. Bitmaps take up a remarkable amount of heap space. Developers often look at the size of a JPEG file and think that "oh, well, that's only a handful of KB", without taking into account:

1. the fact that most image formats, like JPEG and PNG, are compressed, and Android needs the uncompressed image to know what to draw
2. the fact that each pixel may take up several bytes (2 bytes per pixel for `RGB_565`, 3 bytes per pixel for `RGB_888`)
3. what matters is the resolution of the bitmap in its original form, as much (if not more) than the size in which it will be rendered – an 800x480 image displayed in an 80x48 `ImageView` still consumes 800x480 worth of pixel data
4. there are an awful lot of pixels in an image — 800 times 480 is 384,000

Android can make some optimizations, such as only loading in one copy of a Drawable resource no matter how many times you render it. However, in general, each bitmap you load takes a decent sized chunk of your heap, and too many bitmaps means not enough heap. It is not unheard of for an application to have more than half of its heap space tied up in various bitmap images.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Compounding this problem is that bitmap memory, before Honeycomb, was difficult to measure. In the actual Dalvik heap, a Bitmap would need ~80 bytes or so, regardless of image size. The actual pixel data was held in "native heap", the space that a C/C++ program would obtain via calls to `malloc()`. While this space was still subtracted from the available heap space, many diagnostic programs — such as MAT, to be examined in the next chapter — will not know about it. Android 3.0 (code-named "Honeycomb") moved the pixel data into the Dalvik heap, which will improve our ability to find and deal with memory leaks or overuse of bitmaps.

Bitmap Caching

Many Android libraries, like Picasso, offer bitmap caching. Using an existing caching implementation is a lot easier than is rolling your own.

However:

- Make sure that the library is intelligently sizing the cache based upon possible heap space, such as via `getMemoryClass()` as is noted earlier in this answer.
- Where possible, tie your cache to `onTrimMemory()` so that you can flush that cache when appropriate.
- Be careful about using *multiple* libraries, each of which might implement its own cache, that you do not wind up caching too much overall. Each library tends to think that it is The One True Cache for your app and will be oblivious to any other caches — bitmap or otherwise — that you may have in your app. Ideally, the library will have a way for you to set the maximum cache size.

Bitmap Sizing

Sometimes, you do not need a full-size image. For example, if you are showing thumbnails of images in a `ListView`, but only expect to show the full-size image for a few (e.g., rows that the user clicks upon), it is wasteful to load the full-size image for everything in the list.

`BitmapFactory.Options` offers `inSampleSize`, which tells the framework to sample the image as it is loaded, to result in a smaller image. `inSampleSize` of 2 will result in an image that is half the width and half the height; `inSampleSize` of 4 will result in an image that is a quarter the width and a quarter the height; etc. Note that `inSampleSize` is limited to powers of 2 and will round as needed.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you know ahead of time the size of the image, you can calculate an appropriate `inSampleSize` to use. Otherwise, for local content, you can use `BitmapFactory` twice:

- Once with a `BitmapFactory.Options` set with `inJustDecodeBounds` set to `true`, which will merely tell you how big the image is via `outHeight` and `outWidth` on the `BitmapFactory.Options` itself
- Once with a `BitmapFactory.Options` set with `inSampleSize` set to your desired value and `inJustDecodeBounds` set to `false`, to really load the image, but downsampled to consume less memory

This approach does not work well for images being downloaded directly from the Internet, as you do not want to download the image twice, once just to figure out how big it is. Instead, download the image *without* using `BitmapFactory` to a local file, then use `BitmapFactory` to load in the image. If you are electing to use a two-level cache (memory plus disk), you might download the image to the disk cache, for example.

For example, let's look at the `Bitmaps/InSampleSize` sample project, which demonstrates the memory impact (and visual impact) of loading bitmaps at varying sample sizes.

In the `assets/` directory, we have a ~70KB JPEG file of a flower (courtesy of the Wikimedia Project) and a ~50KB PNG of the CommonsWare logo. Both images are 672 pixels square, which makes them relatively large images. These are in assets to ensure that Android will not attempt any sort of density-based conversion of the images, if they were to be in a drawable resource directory.

The `MainActivity` of the project simply loads up a `ViewPager` and attaches it to a `SampleAdapter`:

```
package com.commonsware.android.bitmap.iss;

import android.app.Activity;
import android.os.Bundle;
import android.support.v4.view.PagerAdapter;
import android.support.v4.view.ViewPager;

public class MainActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ViewPager pager=(ViewPager)findViewById(R.id.pager);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    pager.setAdapter(buildAdapter());
  }

  private PagerAdapter buildAdapter() {
    return(new SampleAdapter(this, getFragmentManager()));
  }
}
```

`SampleAdapter`, in turn, populates the `ViewPager` with four instances of a `BitmaFragment`, where we supply the `newInstance()` factory method of `BitmapFragment` with a value of 1, 2, 4, or 8 (`1 << position`), indicating the `inSampleSize` value we want to use for that fragment instance:

```
package com.commonsware.android.bitmap.iss;

import android.app.Fragment;
import android.app.FragmentManager;
import android.content.Context;
import android.support.v13.app.FragmentPagerAdapter;

public class SampleAdapter extends FragmentPagerAdapter {
  Context ctxt=null;

  public SampleAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);
    this.ctxt=ctxt;
  }

  @Override
  public int getCount() {
    return(4);
  }

  @Override
  public Fragment getItem(int position) {
    return(BitmapFragment.newInstance(1 << position));
  }

  @Override
  public String getPageTitle(int position) {
    return(BitmapFragment.getTitle(ctxt, 1 << position));
  }
}
```

`BitmapFragment` then:

- Inflates a layout consisting of four `ImageView` widgets, two at `672dp` square for the "natural" size (scaling only for density), and two at `128dp` square to illustrate how the images appear when constrained to a smaller space:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
      android:id="@+id/byte_count"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center_horizontal"
      android:layout_marginBottom="16dp"
      android:textSize="20sp"
      android:textStyle="bold"/>

    <ImageView
      android:id="@+id/flower_large"
      android:layout_width="672dp"
      android:layout_height="672dp"
      android:layout_gravity="center_horizontal"
      android:layout_marginBottom="16dp"
      android:contentDescription="@string/flower_large"
      android:scaleType="fitCenter"/>

    <ImageView
      android:id="@+id/logo_large"
      android:layout_width="672dp"
      android:layout_height="672dp"
      android:layout_gravity="center_horizontal"
      android:layout_marginBottom="16dp"
      android:contentDescription="@string/logo_large"
      android:scaleType="fitCenter"/>

    <ImageView
      android:id="@+id/flower_small"
      android:layout_width="128dp"
      android:layout_height="128dp"
      android:layout_gravity="center_horizontal"
      android:layout_marginBottom="16dp"
      android:contentDescription="@string/flower_small"
      android:scaleType="fitCenter"/>

    <ImageView
      android:id="@+id/logo_small"
      android:layout_width="128dp"
      android:layout_height="128dp"
      android:layout_gravity="center_horizontal"
      android:contentDescription="@string/logo_small"
      android:scaleType="fitCenter"/>
  </LinearLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
</ScrollView>
```

- Uses a private `load()` method to load the images at the desired `inSampleSize` using a `BitmapFactory.Options` object
- Pours the images each into two `ImageView` widgets, one large and one small
- Updates some `TextView` widgets in the fragment to show how much memory those images are consuming

```
package com.commonsware.android.bitmap.iss;

import android.annotation.TargetApi;
import android.app.Fragment;
import android.content.Context;
import android.content.res.AssetManager;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.Build;
import android.os.Bundle;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.TextView;
import java.io.IOException;

public class BitmapFragment extends Fragment {
  private static final String KEY_SAMPLE_SIZE="inSampleSize";
  private AssetManager assets=null;

  static BitmapFragment newInstance(int inSampleSize) {
    BitmapFragment frag=new BitmapFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_SAMPLE_SIZE, inSampleSize);
    frag.setArguments(args);

    return(frag);
  }

  static String getTitle(Context ctxt, int inSampleSize) {
    return(String.format(ctxt.getString(R.string.title), inSampleSize));
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.sample, container, false);
    int inSampleSize=getArguments().getInt(KEY_SAMPLE_SIZE, 1);
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    try {
      Bitmap flower=
          load("Tibouchina_urvilleana_flower_ja.jpg", inSampleSize);
      Bitmap logo=load("square.png", inSampleSize);

      ImageView iv=(ImageView)result.findViewById(R.id.flower_large);

      iv.setImageBitmap(flower);
      iv=(ImageView)result.findViewById(R.id.flower_small);
      iv.setImageBitmap(flower);
      iv=(ImageView)result.findViewById(R.id.logo_large);
      iv.setImageBitmap(logo);
      iv=(ImageView)result.findViewById(R.id.logo_small);
      iv.setImageBitmap(logo);

      TextView tv=(TextView)result.findViewById(R.id.byte_count);

      tv.setText(String.valueOf(byteCount(flower)));
    }
    catch (IOException e) {
      Log.e(getClass().getSimpleName(), "Exception loading bitmap", e);
    }

    return(result);
  }

  private Bitmap load(String path, int inSampleSize) throws IOException {
    BitmapFactory.Options opts=new BitmapFactory.Options();

    opts.inSampleSize=inSampleSize;

    return(BitmapFactory.decodeStream(assets().open(path), null, opts));
  }

  private AssetManager assets() {
    if (assets == null) {
      assets=getActivity().getResources().getAssets();
    }

    return(assets);
  }

  @TargetApi(Build.VERSION_CODES.KITKAT)
  private int byteCount(Bitmap b) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
      return(b.getAllocationByteCount());
    }

    return(b.getByteCount());
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If you run this on a device, you will see the images at the various sample sizes, one sample size per page of the `ViewPager`. While the quality of the loaded images decreases as `inSampleSize` increases, the smaller `ImageView` widgets are still usable for the flower JPEG, though the line-art PNG suffers.

NOTE: The following screenshots will themselves be modified as part of the publishing process of the book and are here only for illustration purposes. You will want to run the demo and see the results first-hand.

Figure 777: InSampleSize Demo, on an LG Pad 8.3, inSampleSize = 1, Flower JPEG, Full Size

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

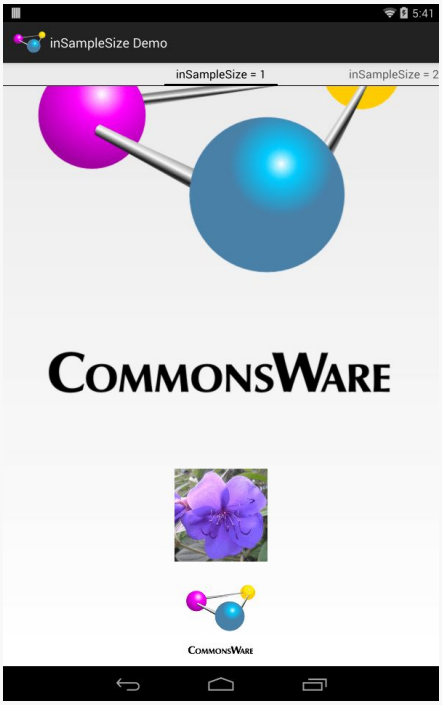


Figure 778: InSampleSize Demo, on an LG Pad 8.3, inSampleSize = 1, CW Logo PNG (Full Size) and Smaller Sizes

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

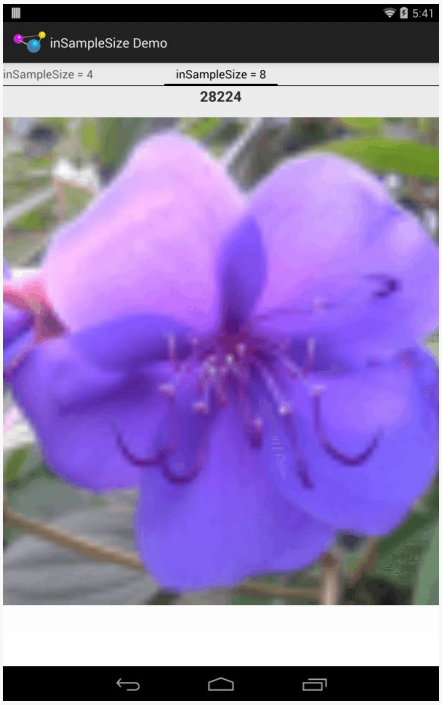


Figure 779: InSampleSize Demo, on an LG Pad 8.3, inSampleSize = 8, Flower JPEG, Full Size

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

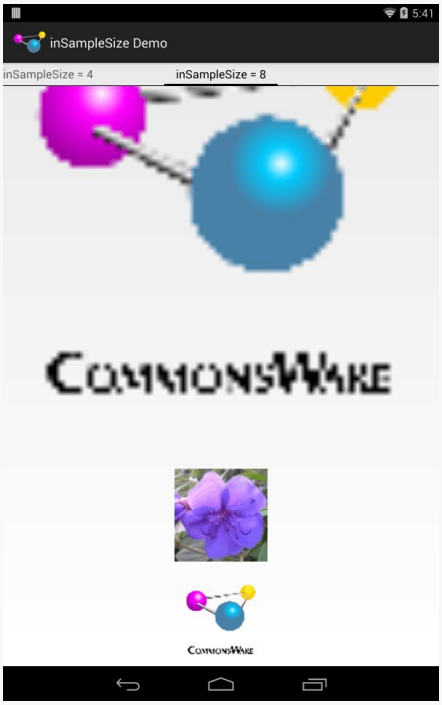


Figure 780: InSampleSize Demo, on an LG Pad 8.3, inSampleSize = 8, CW Logo PNG (Full Size) and Smaller Sizes

The key, though, is the reduced memory footprint. The images loaded without sampling (`inSampleSize` of 1) take up 1,806,336 bytes of heap space (672 x 672 x 4 bytes per pixel). The `inSampleSize` of 8, by contrast, take up 28,244 bytes of heap space, less than 2% of the original.

You should consider experimenting with `inSampleSize` and determine an appropriate sampling level for the types of images you will receive (photos work better than line art) and the sizes you intend to use them in.

Bitmap Color Space

`BitmapFactory` will load images as `ARGB_8888` by default. That means that each pixel takes up four bytes, one each for the red, green, and blue color channels, plus a byte for the alpha channel (transparency).

However, particularly for thumbnails of photographs, where transparency probably does not exist and the image is small when viewed by the user, four bytes per pixel may be overkill.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Instead, you can set `inPreferredConfig` of the `BitmapFactory.Options` to `RGB_565`, which uses only two bytes (five bits for red, six bits for green, five bits for blue, and no transparency). This will cut your memory consumption for the bitmap in half, with no loss of resolution (as you get with `inSampleSize`).

Bitmap Reuse

If you will be doing a lot of work with bitmaps, particularly bitmaps of the same size, an object pool can be of tremendous help to minimize heap fragmentation. You can reuse the same `Bitmap` over and over again, by supplying it via `inBitmap` in the `BitmapFactory.Options` object. If the `Bitmap` is compatible with what you are looking to decode, it will be reused, rather than have a new `Bitmap` (backed by a new hunk of heap space) be created.

Here, "compatible" means:

- The image is the same bit depth configuration (`ARGB_8888` versus `RGB_565`)
- For API Level 18 and below, the resolution is identical; for API Level 19+, the `inBitmap` resolution is the same as or higher than the bitmap to be loaded

Releasing SQLite Memory

SQLite maintains a "page cache" of loaded pages from your database files. Curiously, it does so on a static basis, not on a per-`SQLiteDatabase` basis. Hence, even after you have closed your databases, you might still be consuming more memory than you need to, due to this cache.

From `onTrimMemory()`, you can call the static `releaseMemory()` method on SQLite, to try to free up some of this memory. This should not cause any database errors, but it may slow down the next few database accesses, as the necessary pages may no longer be cached and may have to be loaded again from disk.

In Too Deep (on the Stack)

Heap, however, is not the only possible source of memory errors. It is also possible to get a `StackOverflowError`, indicating that you have run out of stack space (or possibly that the leading Android developer support resource is down for maintenance).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In stack-based programming languages like Java, each time you call a method, some stack space is consumed. While method parameters are objects that live on the heap, the parameter references are stored on the stack, as is information about the method being invoked. References to local data members to the method or blocks inside of it are also stored on the stack.

Since these references only take up ~4 bytes each, you would think it might take a minor eternity to run out of stack space. However, the main application thread in your Android application has an 8KB stack (at least on older devices), which means you can run out of stack space with only a couple of thousand objects on it.

Even still, it would take hundreds and hundreds of nested method invocations to put a couple of thousand objects onto the stack. In normal programming, you might only encounter this with a runaway bit of recursion, in which case no amount of stack would save you.

However, Android GUIs are fairly stack-driven. You can run out of stack space if your UI becomes too complex. More specifically, you might run out of stack space if your view hierarchy — from the root container of the Android window to the widgets inside of the containers inside of your rows inside of your `ListView` inside of your `TabHost` — gets too deep. A depth of 15 or so makes you very likely to run out of stack space somewhere along the line. So if you get the stack-space exception and the stack trace seems to be all in Android UI rendering code, your view hierarchy is probably too complex. In this part of the book, we will examine how to measure your view hierarchy depth and ways of trying to simplify it.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Finding Memory Leaks with MAT

The Eclipse Memory Analyzer (MAT) is your #1 tool for identifying memory leaks and the culprits behind running out of heap space. Particularly when used with Honeycomb or newer versions of Android, MAT can identify:

1. Who are the major sources of memory consumption, both directly (e.g., bitmaps) or indirectly (e.g., leaked activities holding onto lots of widgets)
2. What is keeping objects in memory unexpectedly, defying standard garbage collection — the way that you leak memory in a managed runtime environment like Dalvik

This chapter will identify how to collect heap data for use with MAT and how to use MAT to make sense of what the heap is trying to tell us about what is going on inside of your app.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate, particularly the chapter on Android's process model. Reading the introductory chapter to this trail might be nice.

Setting Up MAT

MAT is an official Eclipse project, hosted on the Eclipse Web site. It comes in two flavors:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. A plug-in for Eclipse itself, providing a new "Memory Analysis" perspective and related tools
2. A standalone version, running in the Eclipse RCP framework

Developers who use Android Studio, or any non-Eclipse IDE, will need to use the standalone version. Some Eclipse users may prefer the standalone version, because they run into problems when their Eclipse workspaces have too many plugins. Some developers may prefer the integrated version, because two Eclipse-based apps would consume too much RAM.

There is a traditional download link to get the standalone edition. As with other Eclipse plug-ins, you will need to add the MAT update site to Eclipse — for example, in Eclipse Galileo:

1. Choose Help|Install New Software... from the main menu
2. Click the Add... button in the upper-right corner of the dialog, fill in `http://download.eclipse.org/mat/1.1/update-site/` as the Location and whatever name you want, then click OK
3. Choose Memory Analyzer for Eclipse IDE and complete the rest of the new-software wizard

Getting Heap Dumps

The first step to analyzing what is in your heap is to actually get your hands on what is in your heap. This is referred to as creating a "heap dump" — what amounts to a log file containing all your objects and who points to what.

There are multiple ways of obtaining a heap dump, depending on your tools and use cases. Note that you will find some blog post and the like indicating you can create a heap dump via the `adb shell kill` command, but this has been disabled in newer versions of Android.

From DDMS

You can get a heap dump any time you want from DDMS, using either the Android/DDMS view in Android Studio, the DDMS perspective in Eclipse or via the Android Device Monitor.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android Studio

In the Android/DDMS view in Android Studio, when you have selected a process in the list, a "Dump Java Heap" toolbar button will be enabled:

Figure 781: "Dump Java Heap" Toolbar Button in Android Studio

Tapping that, and waiting a few moments, will pop up a dialog where you can specify where you want the heap dump file to be written to. You may wish to use the `.hprof` extension for this file. Also, the dialog will have a checkbox asking if you want to "Convert to standard HPROF format" — please ensure that this checkbox is checked.

Eclipse

In the device-and-process tree (the Devices tool in Eclipse), you will find a toolbar button that looks like a half-empty can with a downward-pointing arrow:

Figure 782: "Dump HPROF File" Toolbar Button Icon

Clicking this — after choosing your desired process — DDMS will create a heap dump for you. However, the process varies at this point, depending on whether you are using the DDMS perspective in Eclipse or standalone DDMS.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Once you click the toolbar button for the heap dump, DDMS will create the dump for you, in a file generated in your development machine's temporary-files directory (e.g., `/tmp`). If you wish to save this dump for some reason, you will want to rename it and move it to some other location.

Android Device Monitor

The process for generating a heap dump from DDMS in the Android Device Monitor goes through the same steps as is described for Eclipse above.

Then, however, you will need to run the `hprof-conv` utility, from the `tools/` directory of your SDK, to convert the heap dump into the format that MAT will use. This is automatic if you use the DDMS perspective in Eclipse or have the "convert" checkbox checked in the save-file dialog in Android Studio.

From Code

Another possibility is to trigger the heap dump yourself from code. The `dumpHprofData()` static method on the `Debug` class (in the `android.os` package) will write out a heap dump to the file you indicate. Since these files can be big, and since you will need to transfer them off the device or emulator, it will be best to specify a path to a file on external storage, which means that your project will need the `WRITE_EXTERNAL_STORAGE` permission.

To view the results in MAT, you will need to transfer the file to your development machine (e.g., DDMS File Manager, `adb pull`, using MTP-mounted external storage on Android 3.0+).

Automating Heap Dumps in Testing

One problem with using `dumpHprofData()` is that there is no logical reason to have that code in your production app. Fortunately, you can use it from a JUnit test suite that uses the Android instrumentation framework. However, the main project, not the test project, is the one that needs `WRITE_EXTERNAL_STORAGE` — with luck, your app needs this permission anyway.

The problem then becomes a matter of figuring out where in the JUnit test suite to call `dumpHprofData()`. One strategy is simply to add it to specific test methods or test cases, if you want to have a dump at specific points. If, however, you want a

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

dump at the end of the complete battery of tests, you will need to create your own test runner.

For example, in the MAT/Spinners sample project, you will find a near-identical clone of the same project from elsewhere in this book. It simply runs through a pathetic little test suite for an app that displays contact data in a `ListView`, driven by a `Spinner` to select what data you want to see.

The augmented version of this project adds an `HprofTestRunner` that will dump the heap at the end of the run:

```
package com.commonsware.android.contacts.spinners;

import java.io.File;
import java.io.IOException;
import android.os.Bundle;
import android.os.Debug;
import android.os.Environment;
import android.test.InstrumentationTestRunner;

public class HprofTestRunner extends InstrumentationTestRunner {
  @Override
  public void finish(int resultCode, Bundle results) {
    try {
      Debug.dumpHprofData(new File(Environment.getExternalStorageDirectory(),
"hprof.dmp").getAbsolutePath());
    }
    catch (IOException e) {
      e.printStackTrace();
    }

    super.finish(resultCode, results);
  }
}
```

To add code at the end of a test run, simply override the `finish()` method, do your work, then chain to the superclass. Here, we create an `hprof.dmp` file out in the root of external storage. Note that the runner does not log to LogCat, which is why this code uses the classic `printStackTrace()` to dump any exceptions to the test runner's own error log.

To use the `HprofTestRunner`, you need to update the `android:name` attribute in the `<instrumentation>` element in your manifest to reference this runner class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- package name must be unique so suffix with "tests" so package loader
doesn't ignore us -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
          package="com.commonsware.android.contacts.spinners.tests"
          android:versionCode="1"
          android:versionName="1.0">
    <!-- We add an application tag here just so that we can indicate that
         this package needs to link against the android.test library,
         which is needed when building test cases. -->
    <application>
        <uses-library android:name="android.test.runner" />
    </application>
    <!--
    This declares that this application uses the instrumentation test runner
targeting
    the package of com.commonsware.android.contacts.spinners.  To run the tests
use the command:
    "adb shell am instrument -w com.commonsware.android.contacts.spinners.tests/
android.test.InstrumentationTestRunner"
    -->
    <instrumentation
android:name="com.commonsware.android.contacts.spinners.HprofTestRunner"

android:targetPackage="com.commonsware.android.contacts.spinners"
                     android:label="Tests for
com.commonsware.android.contacts.spinners"/>
</manifest>
```

Also, in your `build.xml` file for Ant, you will need to add the `test.runner` property, identifying the same class, before the final `<import/>` tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="SpinnersTests" default="help">

    <!-- The local.properties file is created and updated by the 'android' tool.
         It contains the path to the SDK. It should *NOT* be checked into
         Version Control Systems. -->
    <property file="local.properties" />

    <!-- The ant.properties file can be created by you. It is only edited by the
         'android' tool to add properties to it.
         This is the place to change some Ant specific build properties.
         Here are some properties you may want to change/update:

         source.dir
             The name of the source directory. Default is 'src'.
         out.dir
             The name of the output directory. Default is 'bin'.

         For other overridable properties, look at the beginning of the rules
         files in the SDK, at tools/ant/build.xml

         Properties related to the SDK location or the project target should
         be updated using the 'android' tool with the 'update' action.

         This file is an integral part of the build system for your
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
         application and should be checked into Version Control Systems.

         -->
    <property file="ant.properties" />

    <!-- if sdk.dir was not set from one of the property file, then
         get it from the ANDROID_HOME env var.
         This must be done before we load project.properties since
         the proguard config can use sdk.dir -->
    <property environment="env" />
    <condition property="sdk.dir" value="${env.ANDROID_HOME}">
        <isset property="env.ANDROID_HOME" />
    </condition>

    <!-- The project.properties file is created and updated by the 'android'
         tool, as well as ADT.

         This contains project specific properties such as project target, and
library
         dependencies. Lower level build properties are stored in ant.properties
         (or in .classpath for Eclipse projects).

         This file is an integral part of the build system for your
         application and should be checked into Version Control Systems. -->
    <loadproperties srcFile="project.properties" />

    <!-- quick check on sdk.dir -->
    <fail
            message="sdk.dir is missing. Make sure to generate local.properties
using 'android update project' or to inject it through the ANDROID_HOME
environment variable."
            unless="sdk.dir"
    />

    <!--
        Import per project custom build rules if present at the root of the
project.
        This is the place to put custom intermediary targets such as:
            -pre-build
            -pre-compile
            -post-compile (This is typically used for code obfuscation.
                           Compiled code location: ${out.classes.absolute.dir}
                           If this is not done in place, override
${out.dex.input.absolute.dir})
            -post-package
            -post-build
            -pre-clean
    -->
    <import file="custom_rules.xml" optional="true" />

    <!-- Import the actual build file.

         To customize existing targets, there are two options:
         - Customize only one target:
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
            - copy/paste the target into this file, *before* the
              <import> task.
            - customize it to your needs.
        - Customize the whole content of build.xml
            - copy/paste the content of the rules files (minus the top node)
              into this file, replacing the <import> task.
            - customize to your needs.

        **********************
        ****** IMPORTANT ******
        **********************
        In all cases you must update the value of version-tag below to read
'custom' instead of an integer,
        in order to avoid having your file be overridden by tools such as
"android update project"
    -->
    <!-- version-tag: custom -->
    <property name="test.runner"
value="com.commonsware.android.contacts.spinners.HprofTestRunner" />
    <import file="${sdk.dir}/tools/ant/build.xml" />

</project>
```

Then, running the tests via `ant debug install test` will use your runner and will dump the HPROF file at the end of the run. You could also elect to automate retrieving the HPROF file by adding an Ant task that will use `adb pull` to retrieve the file from where it is stored.

If you wish to run your tests through Eclipse, you will need to change the Instrumentation property of your test projects to point to your custom `InstrumentationTestRunner` subclass.

Basic MAT Operation

Once you have MAT installed and you have obtained a heap dump, you can start doing some analysis.

Loading Your Dump

If you used the DDMS perspective in Eclipse to create the heap dump, it should automatically pop you into MAT:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

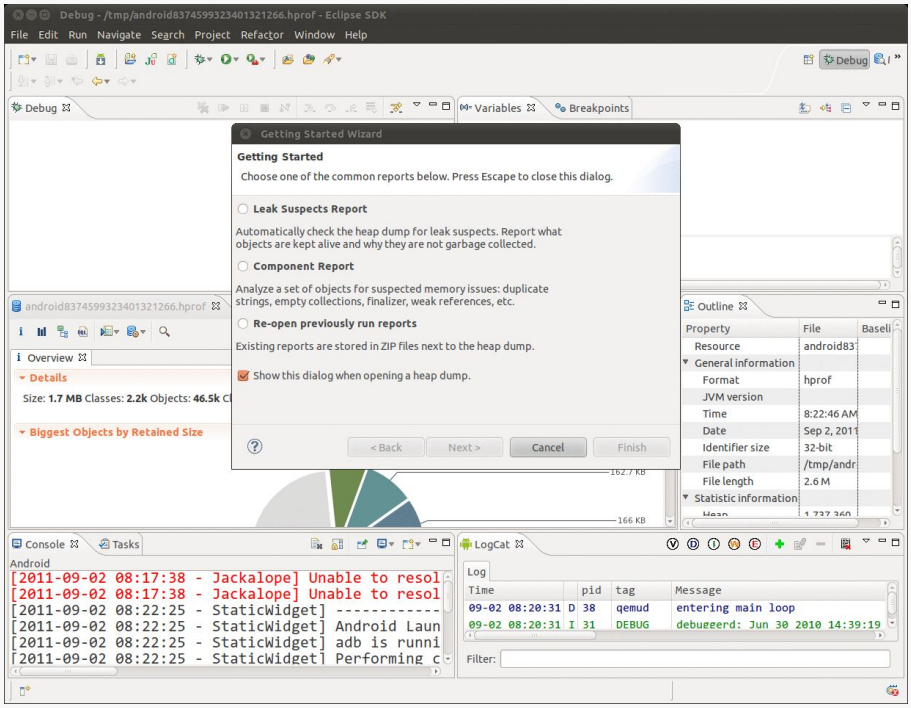


Figure 783: The MAT Eclipse perspective, as initially opened

If you used Android Studio and checked the "convert" checkbox, you can use the File menu in the standalone MAT edition to open up the heap dump that you saved.

If you used the Android Device Monitor or the code-based way of getting a heap dump, after using `hprof-conv` to create a MAT-compatible version of your dump, you can open it using the File|Open Heap Dump... menu from the Eclipse (or standalone MAT) main menu.

The first time you run MAT, you may be presented with the "Getting Started Wizard" (see above screenshot), which you can use or dismiss as you see fit.

The Overview tool gives you, well, an overview of the contents of the heap dump:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

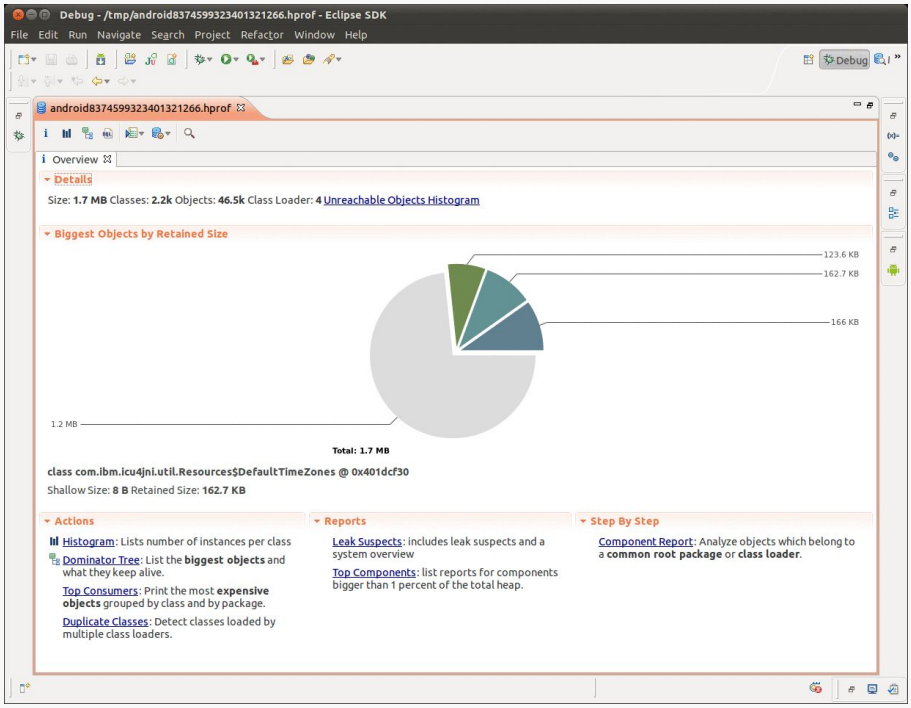


Figure 784: The Overview tool inside the MAT Eclipse perspective

The Overview tool also has links and toolbar buttons to get you to the other major functional areas within MAT.

Finding Your Objects

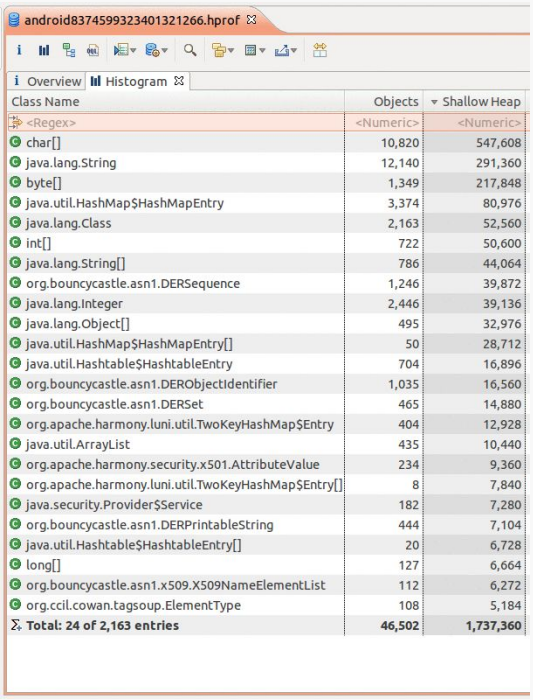
If you want to see if instances of your own classes are being kept in memory despite garbage collection, you can search for objects based upon a regular expression on the fully-qualified class name.

One way to access this is via the Histogram, reachable via a link in the Overview's Actions area or via a toolbar button:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 785: The icon used for the Histogram toolbar button

The histogram initially displays the top culprits in terms of "shallow heap" — the amount of memory those objects hold onto directly:



Class Name	Objects	Shallow Heap
<Regex>	<Numeric>	<Numeric>
char[]	10,820	547,608
java.lang.String	12,140	291,360
byte[]	1,349	217,848
java.util.HashMap$HashMapEntry	3,374	80,976
java.lang.Class	2,163	52,560
int[]	722	50,600
java.lang.String[]	786	44,064
org.bouncycastle.asn1.DERSequence	1,246	39,872
java.lang.Integer	2,446	39,136
java.lang.Object[]	495	32,976
java.util.HashMap$HashMapEntry[]	50	28,712
java.util.Hashtable$HashtableEntry	704	16,896
org.bouncycastle.asn1.DERObjectIdentifier	1,035	16,560
org.bouncycastle.asn1.DERSet	465	14,880
org.apache.harmony.luni.util.TwoKeyHashMap$Entry	404	12,928
java.util.ArrayList	435	10,440
org.apache.harmony.security.x501.AttributeValue	234	9,360
org.apache.harmony.luni.util.TwoKeyHashMap$Entry[]	8	7,840
java.security.Provider$Service	182	7,280
org.bouncycastle.asn1.DERPrintableString	444	7,104
java.util.Hashtable$HashtableEntry[]	20	6,728
long[]	127	6,664
org.bouncycastle.asn1.x509.X509NameElementList	112	6,272
org.ccil.cowan.tagsoup.ElementType	108	5,184
Total: 24 of 2,163 entries	**46,502**	**1,737,360**

Figure 786: The Histogram tab inside the MAT Eclipse perspective

To see what objects of yours might still be in the heap, you can type in a regular expression (e.g., `com.commonsware.*`) in the Regex row at the top of the table, then press [Enter] to view a filtered list of objects based upon that regular expression:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

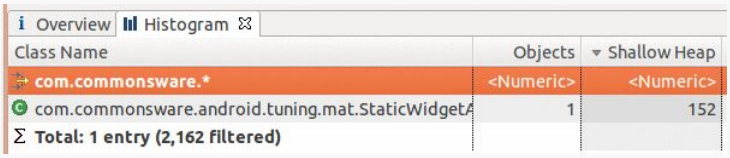


Figure 787: A filtered histogram, showing com.commonsware. objects*

Here, we see one instance of a `com.commonsware` class is still lurking around a heap dump.

Getting Back to Your Roots

However, just because we see an object in MAT does not necessarily mean that it has been leaked. For example, this is an activity – just looking at the above screenshot does not indicate whether that activity was in the foreground, was in the background for normal reasons, or is actually leaked.

To help determine what is keeping the object in memory, you will need to trace back to the "GC roots" — the objects that are preventing our activity from being garbage collected.

To do this, you will right-click over the object in question and choose the "GC Roots" context menu choice (in the Histogram, it is "Merge Shortest Paths to GC Roots"). This will usually bring up a flyout sub-menu where you can further constrain what is reported as a root:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

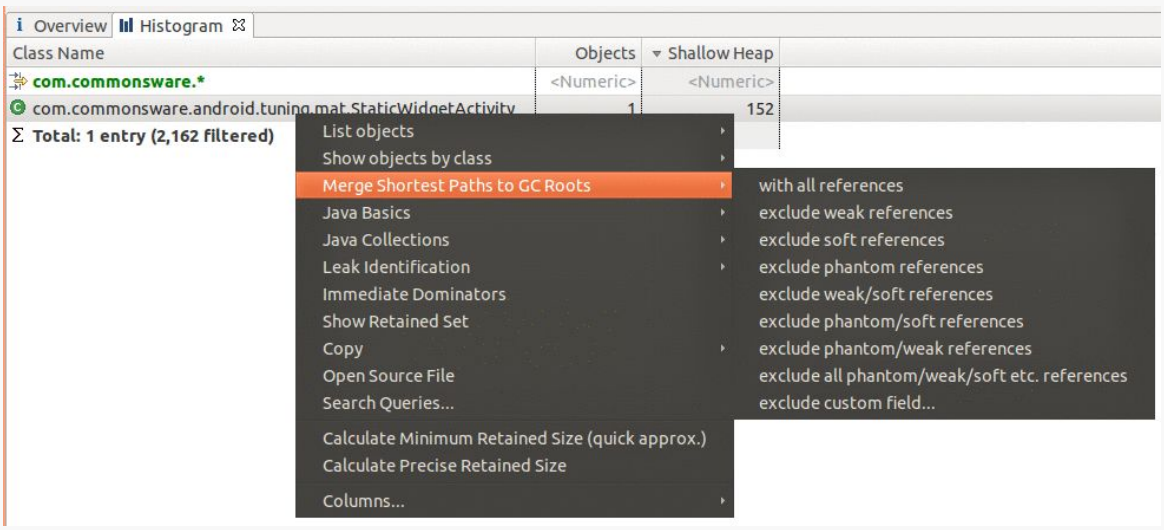


Figure 788: A filtered histogram, showing com.commonsware. objects*

The big filters are for "soft references" and "weak references". These refer to the `SoftReference` and `WeakReference` classes in Java, respectively. Both are ways to hold onto an object yet still allow it to be garbage collected when needed. The big difference is that an object only referenced by `WeakReference` objects can be garbage collected immediately, while an object referenced only by `SoftReference` objects (or a mix of `SoftReference` and `WeakReference` objects) should be kept around until the Dalvik VM is low on memory. Usually, you can ignore weak references, as those just indicate objects that the garbage collector has not quite detected are eligible for reclamation. Whether you want to also filter out soft references would depend a bit on the objects in question — for example, if you are using `SoftReference` with a cache, you might filter out soft references as well to confirm that nothing *other than* your cache is holding onto these objects.

Filtering out weak references (or whatever) brings up another tab containing the GC roots preventing our activity from being garbage collected:

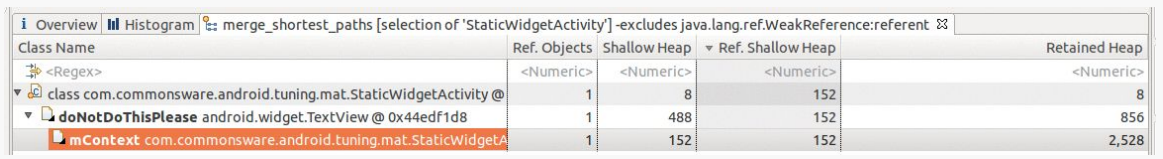


Figure 789: The GC roots holding onto an activity

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This is showing that the *class* for our activity has a data member (`doNotDoThisPlease`) that has a `View`, and that in turn is holding onto our activity via an `mContext` data member. Static data members (i.e., data members of class objects) are classic sources of memory leaks in Java. The Retained Heap column on the far right shows how much memory that individual object (and everything it points to) is keeping around — in this case, about 2.5KB.

Identifying What Else is Floating Around

This helps us find where your own objects are being leaked. What happens if you are leaking other things, though?

One possibility is to examine the rest of the Histogram tab, as it will point out the classes (and primitives) that have the most outstanding instances or hold the most aggregate shallow heap. If you applied a regular expression, you can click on the regular expression and delete it to return to the non-filtered roster. The Histogram tends to report a lot of primitives (e.g., `char` array), and it will take some experience to learn what is standard Android application "noise" and what might represent problems.

Another way to find leaks is to examine the "dominator tree". The term "dominator tree" comes from graph theory — object A "dominates" object B if the only paths to get to B go through A. In MAT, the dominator tree will bubble up those objects whose retained heap — the total memory the object is responsible for, including objects it links to — are high. Or, as MAT describes it, it lists "the biggest objects".

To get to the dominator tree, you can click its link on the Overview tab, or you can click the corresponding toolbar button:

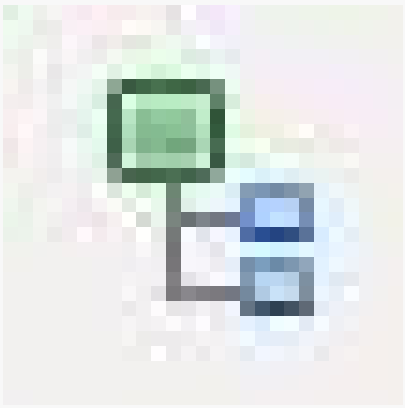


Figure 790: The icon used for the Dominator Tree toolbar button

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This will open up another tab in the same tool, showing “the biggest objects” by retained heap:

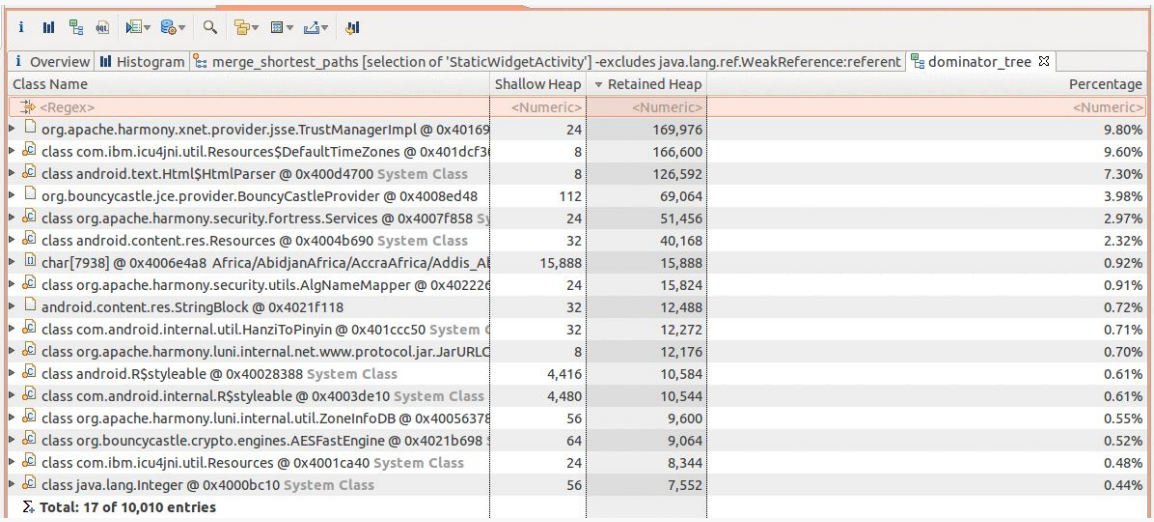


Figure 791: The MAT Dominator Tree tab

You can display more by right-clicking over the Total row at the bottom and choosing "Next 25”.

Here too, the roster will mostly be system objects (e.g., `org.bouncycastle` for the `javax.crypto` implementation). What you would be looking for are objects that you might be interacting with more directly that perhaps you are leaking, such as a Bitmap.

If you find something of interest, right-clicking over the object and choosing “Path to GC Roots” or “Merge Shortest Paths to GC Roots” will help you track down what is holding onto the object, akin to the similar feature in the Histogram.

Some Leaks and Their MAT Analysis

Let’s now take a look at some common leak scenarios in Android and see how we find out whether we have a leak and what is causing it. All of the projects demonstrated below are in the `MAT` directory of the book’s source code.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Widget in Static Data Member

The screen shots from above are mostly taken from the MAT/StaticWidget sample project, where we do something naughty:

```
package com.commonsware.android.tuning.mat;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;

public class StaticWidgetActivity extends Activity {
  @SuppressWarnings("unused")
  static private View doNotDoThisPlease;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    doNotDoThisPlease=findViewById(R.id.make_me_static);
  }
}
```

We take a widget (specifically the auto-generated `TextView`) and put it in a static data member, and never replace it with null.

As a result, even if the user presses BACK to get out of the activity, the static data member holds onto `TextView`, which itself has a reference back to our Activity.

Usually, you will pick this sort of leak up by scanning on your own application's package, as your activity will appear in there. If you are using multiple packages in your application (e.g., yours and a third-party activity), you might need to also check the third-party package to see if any of its objects are being leaked. Whether those leaks are the fault of your code or the third party's own code will vary, of course.

Leaked Thread

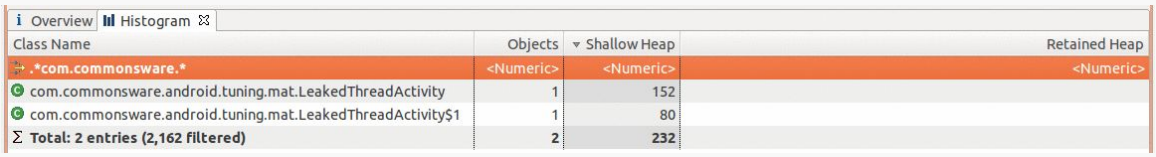
You can see similar results when you leak a thread, such as in the MAT/LeakedThread sample project:

```
package com.commonsware.android.tuning.mat;

import android.app.Activity;
import android.os.Bundle;
import android.os.SystemClock;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
public class LeakedThreadActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    new Thread() {
      public void run() {
        while(true) {
          SystemClock.sleep(100);
        }
      }
    }.start();
  }
}
```

Here, if we filter on `com.commonsware` in the Histogram, we see two entries:



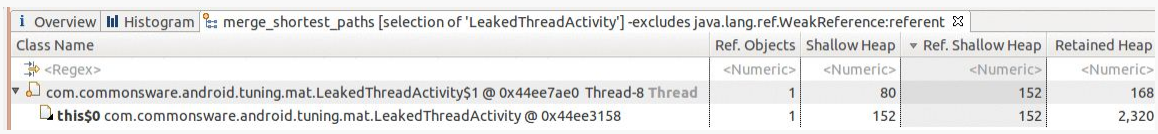
Overview | Histogram

Class Name	Objects	Shallow Heap	Retained Heap
.*com.commonsware.*	<Numeric>	<Numeric>	<Numeric>
com.commonsware.android.tuning.mat.LeakedThreadActivity	1	152	
com.commonsware.android.tuning.mat.LeakedThreadActivity$1	1	80	
Σ Total: 2 entries (2,162 filtered)	2	232	

Figure 792: The LeakedThreadActivity Histogram

As with other places in Java (e.g., stack traces), the `$` syntax in a class name refers to an inner class, and `$1` refers to the first anonymous inner class.

If we look at the GC roots for the activity, we see:



Overview | Histogram | merge_shortest_paths [selection of 'LeakedThreadActivity'] -excludes java.lang.ref.WeakReference:referent

Class Name	Ref. Objects	Shallow Heap	Ref. Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
com.commonsware.android.tuning.mat.LeakedThreadActivity$1 @ 0x44ee7ae0 Thread-8 Thread	1	80	152	168
this$0 com.commonsware.android.tuning.mat.LeakedThreadActivity @ 0x44ee3158	1	152	152	2,320

Figure 793: The GC roots for LeakedThreadActivity

The root is a thread, as denoted by the "Thread" annotation on the end of the root entry. We see that the `Thread` object itself is our `$1` inner class instance, and it holds onto the activity via the implicit reference every non-static inner class has to its outer class instance (`this$0`).

Any running thread will cause anything it can reach to remain in the heap and not get garbage collected. An inner class implementation of the `Thread` — which most code examples will use, in one form or fashion — will leak the outer class instance. Hence, the lessons to be learned here are:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. Leaking threads leaks memory
2. Consider using static inner classes, or separate classes, rather than non-static inner classes, so you do not cause objects to be held onto unnecessarily and unexpectedly

All Sorts of Bugs

Let's now examine the MAT/RandomAppOfCrap sample application. This is a variation on an example from elsewhere in this book, showing using a bound service that connects to a Web service — in this case, the US National Weather Service. In this modified version, a number of leak-related bugs were introduced.

Leaks Via Configuration Changes

The `WeatherDemo` activity implements `onRetainNonConfigurationInstance()`, returning a `State` object. `State` is an inner class of `WeatherDemo`, but not a static inner class.

This is not a good idea.

When you search the Histogram for `com.commonsware` after loading a weather forecast (e.g., run the app and use DDMS to push over a location fix) and rotating the screen, you see that there are two instances of `WeatherDemo` floating around the heap:

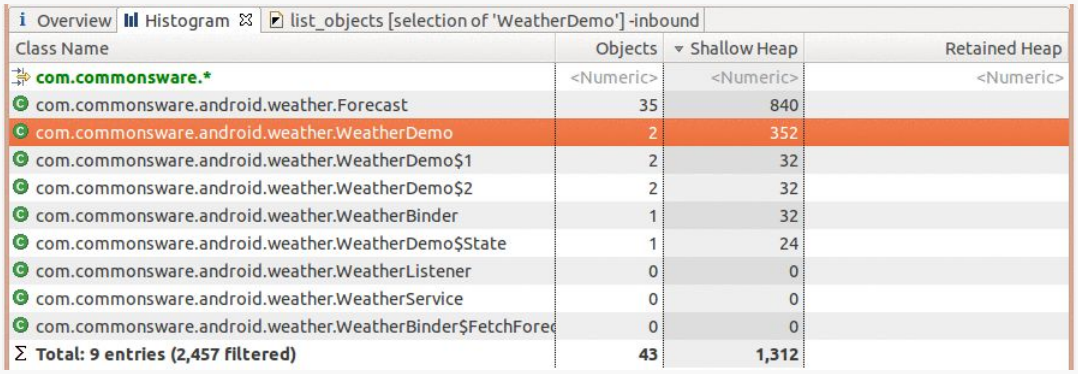
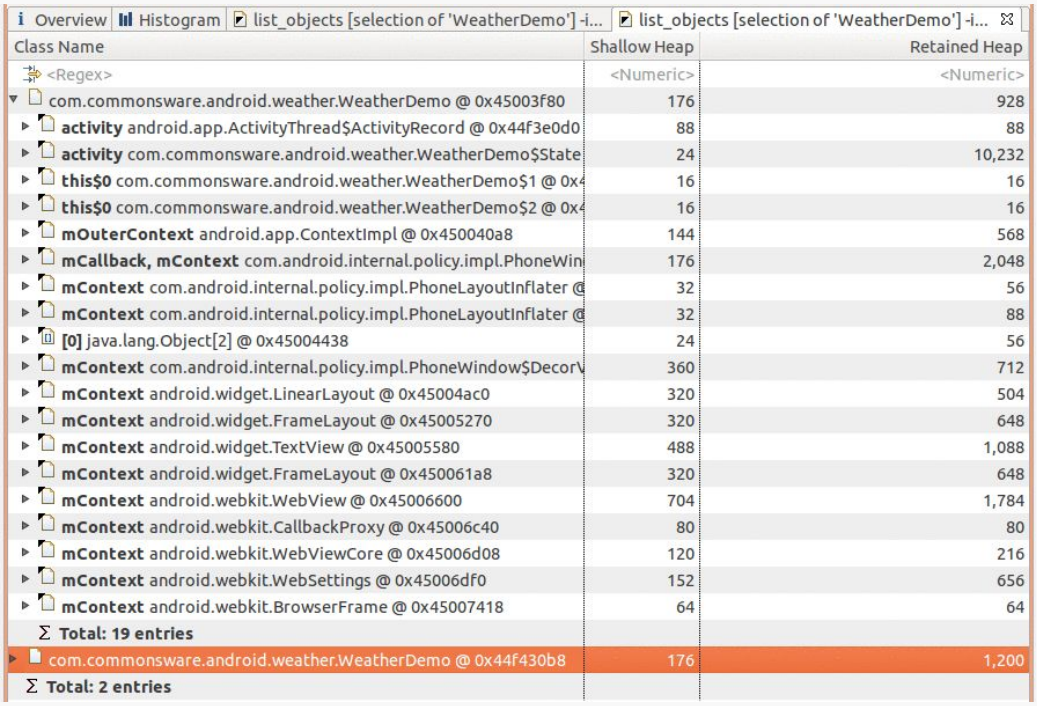


Figure 794: The com.commonsware objects in the RandomAppOfCrap heap

To figure out what those objects are, you can right-click over a class in the Histogram and choose "List Objects" from the context menu. The fly-out sub-menu will let you choose to show incoming references (who points to these objects) or

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

outgoing references (what these objects point to). In this case, showing incoming references will bring up the following:



Overview | Histogram | list_objects [selection of 'WeatherDemo'] -i... | list_objects [selection of 'WeatherDemo'] -i...

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.commonsware.android.weather.WeatherDemo @ 0x45003f80	176	928
activity android.app.ActivityThread$ActivityRecord @ 0x44f3e0d0	88	88
activity com.commonsware.android.weather.WeatherDemo$State	24	10,232
this$0 com.commonsware.android.weather.WeatherDemo$1 @ 0x4	16	16
this$0 com.commonsware.android.weather.WeatherDemo$2 @ 0x4	16	16
mOuterContext android.app.ContextImpl @ 0x450040a8	144	568
mCallback, mContext com.android.internal.policy.impl.PhoneWin	176	2,048
mContext com.android.internal.policy.impl.PhoneLayoutInflater @	32	56
mContext com.android.internal.policy.impl.PhoneLayoutInflater @	32	88
[0] java.lang.Object[2] @ 0x45004438	24	56
mContext com.android.internal.policy.impl.PhoneWindow$DecorV	360	712
mContext android.widget.LinearLayout @ 0x45004ac0	320	504
mContext android.widget.FrameLayout @ 0x45005270	320	648
mContext android.widget.TextView @ 0x45005580	488	1,088
mContext android.widget.FrameLayout @ 0x450061a8	320	648
mContext android.webkit.WebView @ 0x45006600	704	1,784
mContext android.webkit.CallbackProxy @ 0x45006c40	80	80
mContext android.webkit.WebViewCore @ 0x45006d08	120	216
mContext android.webkit.WebSettings @ 0x45006df0	152	656
mContext android.webkit.BrowserFrame @ 0x45007418	64	64
Σ **Total: 19 entries**		
com.commonsware.android.weather.WeatherDemo @ 0x44f430b8	176	1,200
Σ **Total: 2 entries**		

Figure 795: The incoming references for one WeatherDemo instance

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

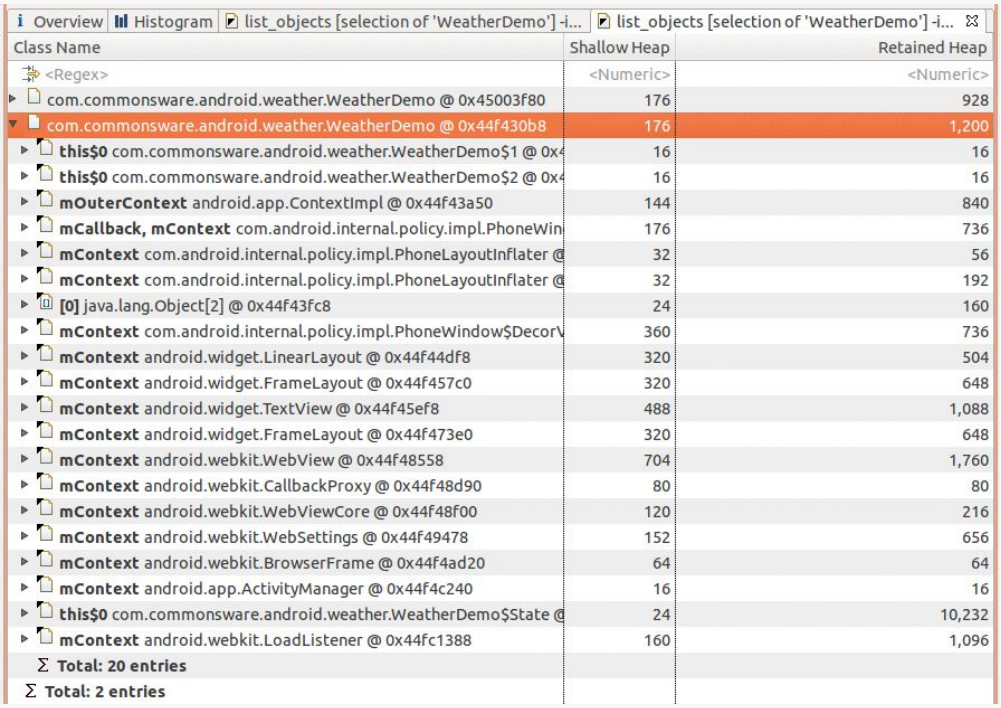


Figure 796: The incoming references for the other WeatherDemo instance

The eight-digit hex numbers shown after the `@` sign are the object identifiers for each of the referred-to objects. You can use this to distinguish which objects are the same.

What you will notice is that both `WeatherDemo` instances are pointed to by the `State` object. In one, it is referred to by the `activity` data member. In the other, it is referred to by `this$0` — the implicit reference an inner class instance has on the outer class instance. Since both `WeatherDemo` instances hold onto the `State` via the `state` data member, this means that one `WeatherDemo` instance (the foreground one) is holding an indirect reference, via the `State`, to the other now-destroyed `WeatherDemo` instance. This is a leak.

The solution for this would be to use a static inner class for `State`, eliminating the implicit reference and breaking this connection.

Leaks from Unregistered System Listeners

We also see from our filtered Histogram that we have two retained instances of the `$1` inner class. Displaying incoming references to those objects shows us that those are the `LocationListener` objects we are using to get our GPS fixes:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

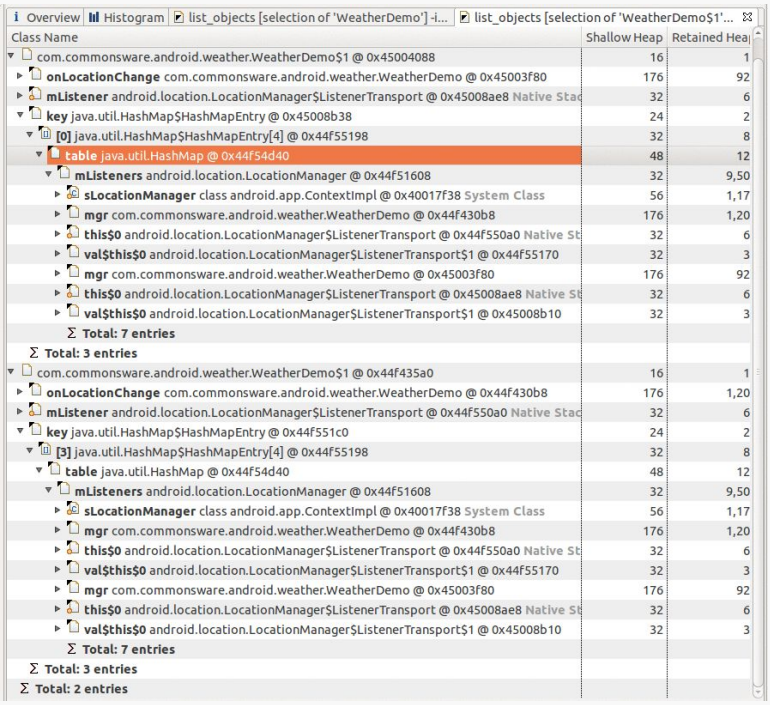


Figure 797: The incoming references for the WeatherDemo$1 instances

Tracing through the incoming references, we see that the `ContextImpl` class holds a static reference to the `LocationManager` system service in our process, and `LocationManager` has an `mListeners` data member which is a list of all registered `LocationListener` instances.

Alas, in `WeatherDemo`, we are registering a `LocationListener` and never unregistering it. Since our `LocationListener` is an inner class, not only is the `LocationListener` itself leaked, but it prevents our destroyed `WeatherDemo` object from being garbage collected.

This same pattern can be seen for many of the system services — if you register a listener, you must ensure that you unregister it to prevent leaks.

What MAT Won't Tell You

MAT is not a universal solution. It may not tell you of all possible leaks.

For example, if you bind to a service, the `ServiceConnection` object you create is held onto, indirectly, by the OS itself. That is how you can use the `ServiceConnection` to unbind from the service later on. However, if you examine

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

MAT, you will see no evidence of this, as MAT is limited to examining *your own* process and cannot report about references that are triggered by other processes.

MAT also will not report anything that is part of the native heap (i.e., what you get with a C `malloc()` call) — it only reports on the Dalvik heap. Hence, MAT will not reflect the actual memory consumption of bitmap images on Gingerbread and earlier environments. You may wish to do some testing of your app on Honeycomb, not just for any tablet support you may offer, but to get more complete results from MAT.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Issues with System RAM

Your application heap is your little corner of the system RAM on the device that you focus on in your Java development. However, there are other things that you might do that consume system RAM, such as use the NDK to add C/C++ code to your app. How much system RAM you consume overall will have an impact on user acceptance of your app, as the more RAM you use, the more frequently the user's other apps are terminated to make room for you. And, as a result, the more system RAM you use, the more likely it is that *your* process will be terminated when you are not in the foreground, to free up RAM for other apps. Hence, while system RAM is not something you necessarily think about as often as you do your application heap, it *is* something that you should pay attention to, at least a little bit.

This chapter will explain a bit more about the relationship between your app and system RAM, how you can measure how much system RAM your app is consuming, and how you can reduce that consumption.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate, particularly the chapter on Android's process model. Reading the chapter on issues with the application heap is also a good idea.

Can't We All Just Get Along?

Alas, we have not invented the device with infinite RAM, nor the application that takes zero memory. In fact, our devices have fairly limited RAM (e.g., 1GB), and our apps therefore fight over that memory. That includes both apps that the user runs

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

explicitly (e.g., via the home screen launcher) and apps that run based upon external factors (e.g., the app that receives a GCM push event and uses that trigger to update some data).

The good news is that the user tends to be a bit oblivious to all of the comings and goings of apps. Android keeps process around while it can and terminates them as needed to free up system RAM for other processes, without the user's explicit involvement. Of course, power users might try to employ "task managers" and the like to be more involved in decision-making, but that's something they opted into, not something that was forced upon them, the way that older mobile operating systems like Windows Mobile required.

However, there is a fundamental assumption in Android that apps play fair. The per-process heap limits — and the fact that apps do not necessarily have to use all the way up to those limits — means that a given Android device can power many processes at once. That starts to break down when apps do various things to consume an excessive amount of system RAM, more than what the per-process heap limit would normally constrain them to. Hence, it is a good idea to keep tabs on how much you use of system RAM, so that you can be a "good citizen" and not cause the user undue angst or force them to employ task managers to try to keep you in line.

Contributors to System RAM Consumption

There are many factors that contribute towards your system RAM consumption, including:

- Your heap usage, up to the per-process heap limit, for each process that you are running
- Your native libraries (`.so` files) from the NDK
- The system RAM allocated by that native code, which does not count against your per-process heap usage

In addition, the reporting tools usually allocate a portion of shared RAM to your app. Your app's process is forked from the `zygote` process, which contains the Dalvik runtime environment, framework JAR (for all those `android.*` classes), and related libraries. Your app shares that memory with all other processes forked from the `zygote`. However, to reflect the fact that there is this overhead, your app's share of it (roughly calculated as the amount of shared RAM divided by the number of processes) tends to get added to your memory consumption totals.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Measuring System RAM Consumption: Tools

Figuring out how much RAM your application is using is not easy. Or, as Dianne Hackborn put it:

> Note that memory usage on modern operating systems like Linux is an extremely complicated and difficult to understand area. In fact the chances of you actually correctly interpreting whatever numbers you get is extremely low. (Pretty much every time I look at memory usage numbers with other engineers, there is always a long discussion about what they actually mean that only results in a vague conclusion.)

Fortunately, particularly in Android 4.4, a fair bit of work has gone into trying to help us determine how much our apps impact system RAM.

Process Stats in Settings

On Android 4.4, in Settings > Developer Options, you will find:

> Process Stats: Geeky stats about running processes

(here, “geeky” is presumably used as a term of endearment)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

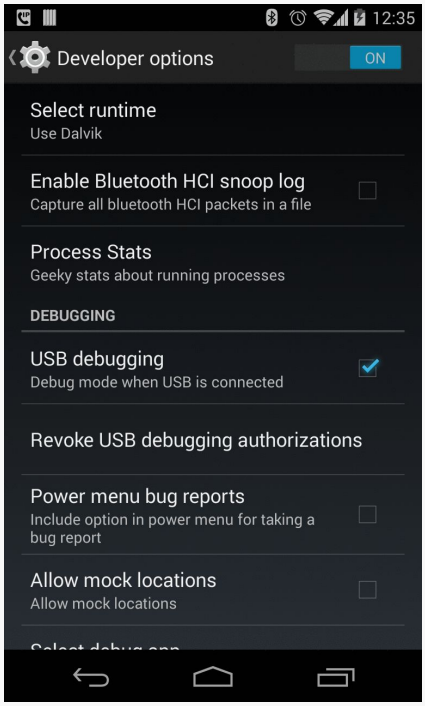


Figure 798: Developer Options in Android 4.4, Showing "Process Stats"

Tapping on that entry brings up a screen that describes the current state of the system, with respect to RAM:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

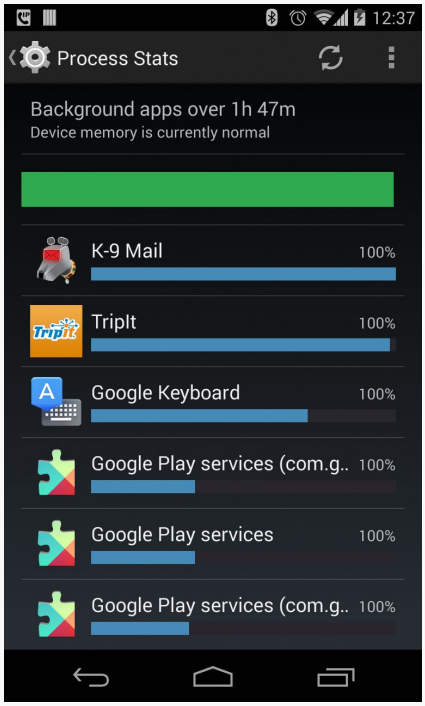


Figure 799: Process Stats in Android 4.4

While it may look simple, this screen, and its child screens, are remarkably complex, particularly once you start playing around with various options from the action bar overflow.

The Summary

At the top, you will see:

- How much history is being reported in this screen ("1h 47m")
- What the current status of the system is with respect to RAM ("currently normal")
- What the status of the system has been over that range of time, as illustrated by the bar, where green is "normal"

The bar is not so much a timeline as a stacked bar chart, where the mix of red and yellow indicates the amount of time the device was in a low-memory state, contrasted with the green "normal" state.

Usually, your device should be "normal" with a mostly-green or completely-green bar.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Roster

The list beneath the summary shows some running processes. What is included in this list depends on what mode Process Stats in running in. The summary indicates that our mode is "Background apps". There are three major categories for apps:

1. Foreground, which includes whatever app is truly in the visible foreground, plus any apps that have a foreground service
2. Background, which is pretty much everything else with a service
3. Cached, which are all apps that still have running processes but do not have a service

By default, Process Stats will show background processes.

Each row in the list shows details for a specific process:

- the application label for the app that owns the process (e.g., "K–9 Mail")
- the percentage of time, for the time period Process Stats is reporting, that this process was running (e.g., "100%")
- the "relative computed memory load" of that process, as is indicated by the blue bar, with longer blue bars indicating greater load

The list is sorted in order of "relative computed memory load".

Many background apps will have a running percentage of 100%, indicating that they have an always-running service. Those with a percentage less than 100% indicate an app that had a service running at the point in time of the Process Stats snapshot that you are examining, but did not necessarily have a service running for the entire timeframe (e.g., periodic `IntentService` doing background work)

Refresh and Duration

There is a "refresh" icon in the action bar that will update the current view to reflect changes since you last opened or refreshed the screen.

How long the timeframe is depends a bit upon device operation and also on the "Duration" entry in the overflow menu:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

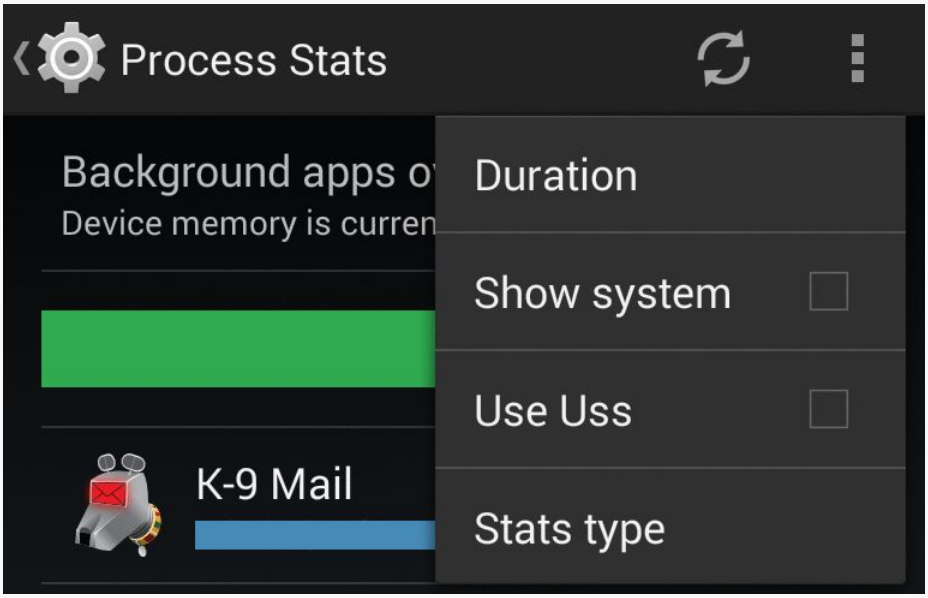


Figure 800: Process Stats Overflow in Android 4.4

Tapping that gives you a roster of available timeframes:

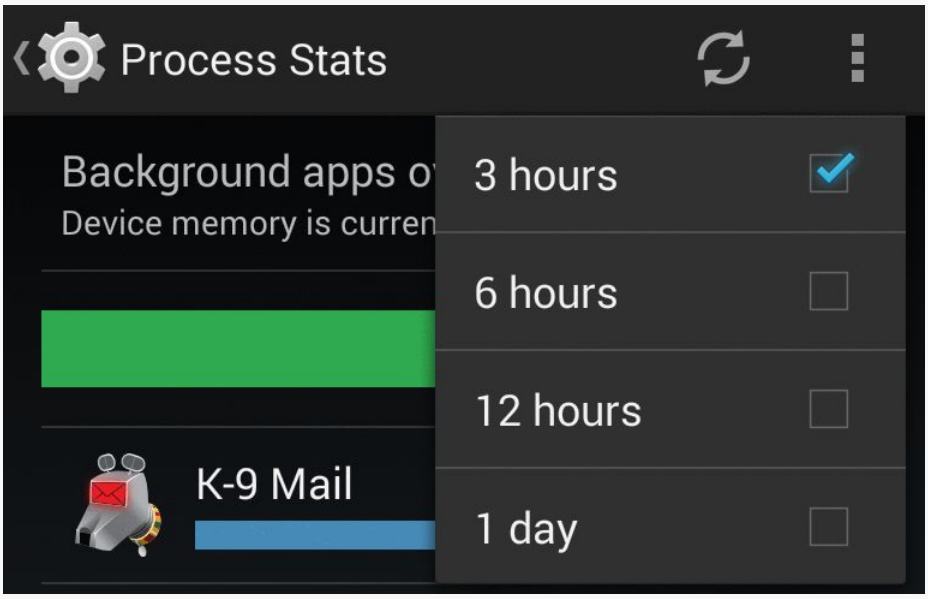


Figure 801: Process Stats Duration Options in Android 4.4

Even though these items render with checkboxes, they function as radio buttons, so whatever you tap on becomes the new duration. Upon making a change, the summary area will reflect the newly-chosen duration. Note that this choice is not

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

persistent, as exiting Process Stats via the BACK button and re-entering it returns you to a three-hour duration.

Controlling What is Shown

If your application is not showing up in the background roster, it may be classified as "foreground" (e.g., if you have a foreground service) or "cached" (if not). The "Stats type" overflow option will let you toggle between these categories, to see what processes are reported in each.

Note that an app can appear in more than one roster, since the roster is by *process*. For example, at the time of this writing, Evernote appears in the author's Nexus 4 both in "foreground" and in "cached", for separate processes.

As with the duration, the choice of category is not persistent, and you will be returned to the background process roster if you exit Process Stats via the BACK button and later return to it.

Drilling Down Into an App

Tapping on an item in the list will bring up details about that particular app and process:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

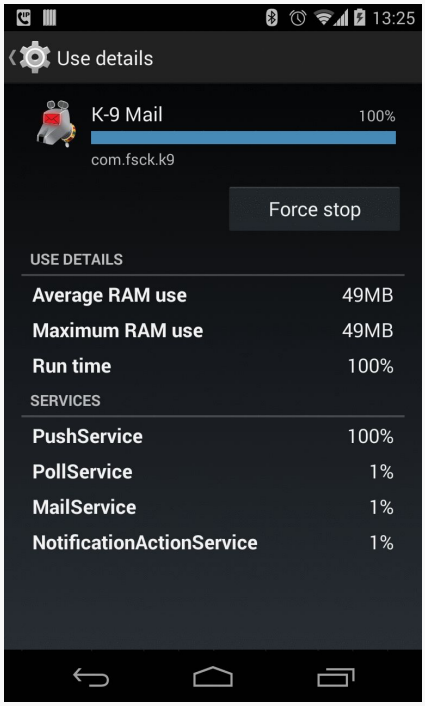


Figure 802: Process Stats Details for K–9 Mail

The "Average RAM use" value shows how much system RAM is attributed to your app. This will include:

- All system RAM used uniquely by your app (e.g., your heap)
- A portion of system RAM shared by your app and others (e.g., the Dalvik runtime)

This is known as "Proportional Set Size" or "PSS" in Linux, and is a common way of coming up with a simple number for the amount of RAM that a particular process is responsible for.

The blue bar is based off of this average RAM use (or PSS) value, multiplied by the percentage of the time that the process was running.

The "Maximum RAM use" value is the highest PSS associated with your process during the period under examination.

Also listed are the services of your app and the percentage of time that they were in a running state. Everlasting services will show up as 100%, while transient services (e.g., `IntentService`) will show up with a much smaller percentage.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

How You Want Your App to Appear

Ideally:

- Your app spends most of its time in the "cached" process list, not the "background" or "foreground" list
- Your app has a low percentage of time spent running
- Your app has a low "relative computed memory load" and, ideally, a low "Average RAM use" value

The better you are in these areas, the more likely it is that you are not seriously impacting system RAM.

procstats

The data that powers the Process Stats screen in Settings is also available as human-readable text output, using the `adb shell dumpsys procstats` command, against your Android 4.4 device or emulator.

Running that command will give you three blocks of information:

- Process memory usage, aggregated over the last 24 hours
- Process memory usage, aggregated over the last 3 hours
- A snapshot of the current memory usage, at the time you ran the command

This can be a long report, even for just one of those blocks. For example, here is the last–3-hours block, run on the author's personal Nexus 4:

```
AGGREGATED OVER LAST 3 HOURS:
  * com.android.bluetooth / 1002:
           TOTAL: 100% (7.8MB-7.8MB-7.8MB/7.0MB-7.0MB-7.0MB over 2)
          Imp Fg: 100% (7.8MB-7.8MB-7.8MB/7.0MB-7.0MB-7.0MB over 2)
  * com.csipsimple:sipStack / u0a135:
           TOTAL: 100% (10MB-10MB-10MB/9.2MB-9.2MB-9.2MB over 1)
          Imp Fg: 99% (10MB-10MB-10MB/9.2MB-9.2MB-9.2MB over 1)
         Service: 1.2%
  * system / 1000:
           TOTAL: 100% (56MB-60MB-63MB/51MB-55MB-58MB over 2)
      Persistent: 100% (56MB-60MB-63MB/51MB-55MB-58MB over 2)
  * com.android.nfc / 1027:
           TOTAL: 100% (6.3MB-6.3MB-6.3MB/5.4MB-5.5MB-5.5MB over 2)
      Persistent: 100% (6.3MB-6.3MB-6.3MB/5.4MB-5.5MB-5.5MB over 2)
  * tunein.player.pro / u0a97:
           TOTAL: 100% (8.2MB-8.2MB-8.2MB/6.9MB-6.9MB-6.9MB over 3)
         Service: 100% (8.2MB-8.2MB-8.2MB/6.9MB-6.9MB-6.9MB over 3)
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
* android.process.acore / u0a0:
        TOTAL: 100% (15MB-15MB-15MB/14MB-14MB-14MB over 1)
       Imp Fg: 0.00%
      Service: 100% (15MB-15MB-15MB/14MB-14MB-14MB over 1)
* com.google.android.gms / u0a23:
        TOTAL: 100% (16MB-16MB-16MB/14MB-14MB-14MB over 2)
      Service: 100% (16MB-16MB-16MB/14MB-14MB-14MB over 2)
* com.espn.radio:com.urbanairship.process / u0a142:
        TOTAL: 100% (6.6MB-6.6MB-6.6MB/5.3MB-5.3MB-5.3MB over 3)
      Service: 100% (6.6MB-6.6MB-6.6MB/5.3MB-5.3MB-5.3MB over 3)
* com.android.launcher / u0a35:
        TOTAL: 100% (73MB-73MB-73MB/69MB-69MB-69MB over 8)
          Top: 100% (73MB-73MB-73MB/69MB-69MB-69MB over 8)
* com.android.systemui / u0a116:
        TOTAL: 100% (38MB-39MB-41MB/35MB-37MB-38MB over 2)
   Persistent: 100% (38MB-39MB-41MB/35MB-37MB-38MB over 2)
* com.android.phone / 1001:
        TOTAL: 100% (26MB-26MB-26MB/25MB-25MB-25MB over 2)
   Persistent: 100% (26MB-26MB-26MB/25MB-25MB-25MB over 2)
* com.tripit / u0a85:
        TOTAL: 100% (44MB-44MB-44MB/41MB-41MB-41MB over 1)
       Imp Fg: 0.72%
      Service: 99% (44MB-44MB-44MB/41MB-41MB-41MB over 1)
* com.google.process.location / u0a23:
        TOTAL: 100% (17MB-17MB-17MB/14MB-14MB-14MB over 2)
       Imp Fg: 100% (17MB-17MB-17MB/14MB-14MB-14MB over 2)
* com.google.android.inputmethod.latin / u0a34:
        TOTAL: 100% (37MB-37MB-37MB/36MB-36MB-36MB over 2)
       Imp Fg: 100% (37MB-37MB-37MB/36MB-36MB-36MB over 2)
* com.google.process.gapps / u0a23:
        TOTAL: 100% (16MB-16MB-16MB/14MB-14MB-14MB over 2)
      Service: 100% (16MB-16MB-16MB/14MB-14MB-14MB over 2)
* com.fsck.k9 / u0a128:
        TOTAL: 100% (50MB-50MB-50MB/47MB-47MB-47MB over 2)
      Service: 100% (50MB-50MB-50MB/47MB-47MB-47MB over 2)
* com.rememberthemilk.MobileRTM / u0a89:
        TOTAL: 17%
       Imp Fg: 8.6%
      Service: 8.6%
     Receiver: 0.11%
     (Cached): 83% (36MB-36MB-37MB/34MB-35MB-35MB over 9)
* android.process.media / u0a15:
        TOTAL: 8.9% (5.4MB-5.4MB-5.4MB/4.6MB-4.6MB-4.6MB over 1)
      Service: 8.9% (5.4MB-5.4MB-5.4MB/4.6MB-4.6MB-4.6MB over 1)
     Receiver: 0.01%
     (Cached): 91%
* com.google.android.apps.maps / u0a39:
        TOTAL: 4.2%
      Service: 4.2%
     (Cached): 96% (121MB-121MB-121MB/105MB-105MB-106MB over 4)
* com.google.android.apps.genie.geniewidget / u0a21:
        TOTAL: 3.9% (5.6MB-5.6MB-5.6MB/4.7MB-4.7MB-4.7MB over 1)
      Service: 3.9% (5.6MB-5.6MB-5.6MB/4.7MB-4.7MB-4.7MB over 1)
     Receiver: 0.00%
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      (Cached): 96% (5.7MB-5.7MB-5.7MB/4.8MB-4.8MB-4.8MB over 1)
* com.google.android.tts / u0a29:
         TOTAL: 1.7%
       Service: 1.7%
      (Cached): 20% (24MB-24MB-24MB/23MB-23MB-23MB over 2)
* com.evernote / u0a86:
         TOTAL: 0.41%
        Imp Bg: 0.31%
       Service: 0.10%
      Receiver: 0.00%
      (Cached): 100% (15MB-15MB-16MB/14MB-14MB-14MB over 4)
* org.mozilla.firefox / u0a100:
         TOTAL: 0.39%
       Service: 0.39%
      (Cached): 100% (4.4MB-6.2MB-7.4MB/3.5MB-5.2MB-6.4MB over 5)
* com.csipsimple / u0a135:
         TOTAL: 0.18%
        Imp Fg: 0.02%
       Service: 0.14%
      Receiver: 0.02%
      (Cached): 100% (4.2MB-4.2MB-4.2MB/3.2MB-3.2MB-3.2MB over 8)
* com.stackexchange.marvin / u0a154:
         TOTAL: 0.17%
       Service: 0.17%
      Receiver: 0.00%
      (Cached): 100% (32MB-32MB-32MB/30MB-30MB-30MB over 2)
* com.google.android.youtube / u0a67:
         TOTAL: 0.12%
       Service: 0.01%
      Receiver: 0.11%
      (Cached): 8.9% (9.3MB-9.3MB-9.3MB/8.0MB-8.0MB-8.0MB over 1)
* com.guywmustang.silentwidget / u0a78:
         TOTAL: 0.05%
       Service: 0.05%
      Receiver: 0.00%
      (Cached): 100% (3.3MB-3.3MB-3.4MB/2.7MB-2.7MB-2.7MB over 4)
* com.google.android.deskclock / u0a14:
         TOTAL: 0.03%
      Receiver: 0.03%
      (Cached): 100% (4.0MB-4.0MB-4.0MB/3.1MB-3.1MB-3.1MB over 2)
* com.google.android.gallery3d / u0a20:
         TOTAL: 0.03%
      Receiver: 0.03%
      (Cached): 9.0% (6.2MB-6.2MB-6.2MB/5.3MB-5.3MB-5.3MB over 1)
* com.szyk.myheart / u0a130:
      (Cached): 100% (35MB-35MB-35MB/31MB-31MB-31MB over 2)
* org.wikipedia / u0a98:
      (Cached): 100% (22MB-22MB-22MB/18MB-18MB-18MB over 2)
* com.android.mms / u0a41:
      (Cached): 100% (23MB-23MB-23MB/21MB-21MB-21MB over 2)
* nz.co.softwarex.hundredpushupsfree / u0a168:
      (Cached): 100% (23MB-23MB-23MB/20MB-20MB-20MB over 2)
* com.commonsware.books.android / u0a148:
      (Cached): 100% (18MB-18MB-18MB/15MB-15MB-15MB over 2)
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  * com.android.providers.calendar / u0a7:
        (Cached): 100% (3.6MB-3.6MB-3.6MB/2.8MB-2.8MB-2.8MB over 2)
  * com.google.android.calendar / u0a6:
        (Cached): 100% (4.7MB-4.7MB-4.7MB/3.8MB-3.8MB-3.8MB over 2)

Run time Stats:
  SOff/Norm: +12m36s333ms
  SOn /Norm: +1m11s367ms
      TOTAL: +13m47s700ms

          Start time: 2014-04-12 06:01:36
  Total elapsed time: +3h54m32s538ms (partial) libdvm.so chromeview
```

Unfortunately, it is rather cryptic and rather long.

There are various command-line switches you can add to help manage the output. Use the `-h` switch to see the full roster. Some notable options:

- `-csv` switches the output to be in CSV format, for importing into a spreadsheet or running through an analysis tool, with other switches (e.g., `-csv-proc`) to control what is included in the CSV output
- `--current` will only report the current snapshot
- `--hours NNN` will only report the aggregate over the stated number of hours
- `--full-details` provides a somewhat more documented report, at the cost of greatly increasing its verbosity

Also, including a package name (e.g., `com.commonsware.android.sample`) at the end of the command line will constrain the output to solely that package, which is useful if you are only looking to examine your own app's data.

The numbers in parentheses (e.g., `(4.7MB-4.7MB-4.7MB/3.8MB-3.8MB-3.8MB over 2)`) report:

- the minimum proportional set size (PSS) seen
- the average PSS seen
- the maximum PSS seen
- the minimum unique set size (USS) seen, where this is the amount of memory consumed by your app that is *not* shared with other processes (i.e., it is how much memory that would be freed if your process were terminated)
- the average USS seen
- the maximum USS seen
- the number of samples taken of the memory during the timeframe being analyzed

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The listing also shows the percentage of time your process was in various states (e.g., cached vs. service vs. "important foreground")

meminfo

Older devices that do not support `procrank` can support `meminfo`, accessed via `adb shell dumpsys meminfo`. Run as-is, it will generate a report of all processes and their PSS, plus the same roster broken down into various process categories (e.g., foreground, cached), and other summary data. The report for the same Nexus 4 that generated the `procrank` shown earlier in this chapter is:

```
Applications Memory Usage (kB):
Uptime: 95955008 Realtime: 788076654

Total PSS by process:
   120505 kB: com.google.android.apps.maps (pid 17490 / activities)
    74627 kB: com.android.launcher (pid 1696 / activities)
    62422 kB: system (pid 1366)
    57757 kB: surfaceflinger (pid 968)
    51706 kB: com.fsck.k9 (pid 2937 / activities)
    44725 kB: com.tripit (pid 2414 / activities)
    41642 kB: com.android.systemui (pid 1498 / activities)
    38546 kB: com.google.android.inputmethod.latin (pid 1635)
    36640 kB: com.rememberthemilk.MobileRTM (pid 12255 / activities)
    35518 kB: com.szyk.myheart (pid 24618 / activities)
    32588 kB: com.stackexchange.marvin (pid 28230 / activities)
    27128 kB: com.android.phone (pid 1667)
    23641 kB: com.android.mms (pid 15197 / activities)
    23236 kB: nz.co.softwarex.hundredpushupsfree (pid 20599 / activities)
    22483 kB: org.wikipedia (pid 11895 / activities)
    18044 kB: com.commonsware.books.android (pid 12036 / activities)
    16968 kB: com.google.process.location (pid 1775)
    16895 kB: com.google.android.gms (pid 1651)
    16521 kB: com.google.process.gapps (pid 1803)
    15822 kB: com.evernote (pid 26284)
    15219 kB: android.process.acore (pid 11926)
    11336 kB: zygote (pid 969)
    10694 kB: com.csipsimple:sipStack (pid 25539)
     9575 kB: com.google.android.youtube (pid 31932)
     8390 kB: tunein.player.pro (pid 2699)
     7860 kB: com.android.bluetooth (pid 5513)
     7669 kB: org.mozilla.firefox (pid 20839)
     6786 kB: com.espn.radio:com.urbanairship.process (pid 2526)
     6719 kB: mediaserver (pid 971)
     6599 kB: com.google.android.gallery3d (pid 31894)
     6493 kB: com.android.nfc (pid 1681)
     5916 kB: com.google.android.apps.genie.geniewidget (pid 22781)
     5677 kB: android.process.media (pid 25240)
     4308 kB: com.csipsimple (pid 28166)
     4145 kB: com.google.android.deskclock (pid 12379)
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
     3472 kB: com.guywmustang.silentwidget (pid 14616)
     3349 kB: rild (pid 967)
     2447 kB: drmserver (pid 970)
     1972 kB: ks (pid 585)
     1876 kB: netd (pid 965)
     1282 kB: wpa_supplicant (pid 26091)
     1217 kB: mm-qcamera-daemon (pid 982)
     1116 kB: sdcard (pid 981)
      618 kB: sensors.qcom (pid 979)
      577 kB: netmgrd (pid 976)
      500 kB: vold (pid 163)
      486 kB: bridgemgrd (pid 974)
      476 kB: thermald (pid 977)
      462 kB: keystore (pid 973)
      439 kB: /init (pid 1)
      375 kB: qmuxd (pid 975)
      262 kB: ueventd (pid 139)
      230 kB: dhcpcd (pid 15630)
      214 kB: qseecomd (pid 1022)
      212 kB: adbd (pid 961)
      210 kB: installd (pid 972)
      189 kB: mpdecision (pid 978)
      181 kB: rmt_storage (pid 164)
      176 kB: dumpsys (pid 489)
      169 kB: qcks (pid 165)
      149 kB: debuggerd (pid 966)
      140 kB: healthd (pid 161)
      135 kB: efsks (pid 569)
      115 kB: servicemanager (pid 162)
      111 kB: qseecomd (pid 986)

Total PSS by OOM adjustment:
    95497 kB: Native
                57757 kB: surfaceflinger (pid 968)
                11336 kB: zygote (pid 969)
                 6719 kB: mediaserver (pid 971)
                 3349 kB: rild (pid 967)
                 2447 kB: drmserver (pid 970)
                 1972 kB: ks (pid 585)
                 1876 kB: netd (pid 965)
                 1282 kB: wpa_supplicant (pid 26091)
                 1217 kB: mm-qcamera-daemon (pid 982)
                 1116 kB: sdcard (pid 981)
                  618 kB: sensors.qcom (pid 979)
                  577 kB: netmgrd (pid 976)
                  500 kB: vold (pid 163)
                  486 kB: bridgemgrd (pid 974)
                  476 kB: thermald (pid 977)
                  462 kB: keystore (pid 973)
                  439 kB: /init (pid 1)
                  375 kB: qmuxd (pid 975)
                  262 kB: ueventd (pid 139)
                  230 kB: dhcpcd (pid 15630)
                  214 kB: qseecomd (pid 1022)
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                212 kB: adbd (pid 961)
                210 kB: installd (pid 972)
                189 kB: mpdecision (pid 978)
                181 kB: rmt_storage (pid 164)
                176 kB: dumpsys (pid 489)
                169 kB: qcks (pid 165)
                149 kB: debuggerd (pid 966)
                140 kB: healthd (pid 161)
                135 kB: efsks (pid 569)
                115 kB: servicemanager (pid 162)
                111 kB: qseecomd (pid 986)
    62422 kB: System
               62422 kB: system (pid 1366)
    75263 kB: Persistent
               41642 kB: com.android.systemui (pid 1498 / activities)
               27128 kB: com.android.phone (pid 1667)
                6493 kB: com.android.nfc (pid 1681)
    74627 kB: Foreground
               74627 kB: com.android.launcher (pid 1696 / activities)
    63374 kB: Visible
               38546 kB: com.google.android.inputmethod.latin (pid 1635)
               16968 kB: com.google.process.location (pid 1775)
                7860 kB: com.android.bluetooth (pid 5513)
    10694 kB: Perceptible
               10694 kB: com.csipsimple:sipStack (pid 25539)
     5677 kB: A Services
                5677 kB: android.process.media (pid 25240)
    51706 kB: Previous
               51706 kB: com.fsck.k9 (pid 2937 / activities)
    76422 kB: B Services
               44725 kB: com.tripit (pid 2414 / activities)
               16521 kB: com.google.process.gapps (pid 1803)
                8390 kB: tunein.player.pro (pid 2699)
                6786 kB: com.espn.radio:com.urbanairship.process (pid 2526)
   402275 kB: Cached
              120505 kB: com.google.android.apps.maps (pid 17490 / activities)
               36640 kB: com.rememberthemilk.MobileRTM (pid 12255 / activities)
               35518 kB: com.szyk.myheart (pid 24618 / activities)
               32588 kB: com.stackexchange.marvin (pid 28230 / activities)
               23641 kB: com.android.mms (pid 15197 / activities)
               23236 kB: nz.co.softwarex.hundredpushupsfree (pid 20599 /
activities)
               22483 kB: org.wikipedia (pid 11895 / activities)
               18044 kB: com.commonsware.books.android (pid 12036 / activities)
               16895 kB: com.google.android.gms (pid 1651)
               15822 kB: com.evernote (pid 26284)
               15219 kB: android.process.acore (pid 11926)
                9575 kB: com.google.android.youtube (pid 31932)
                7669 kB: org.mozilla.firefox (pid 20839)
                6599 kB: com.google.android.gallery3d (pid 31894)
                5916 kB: com.google.android.apps.genie.geniewidget (pid 22781)
                4308 kB: com.csipsimple (pid 28166)
                4145 kB: com.google.android.deskclock (pid 12379)
                3472 kB: com.guywmustang.silentwidget (pid 14616)
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
Total PSS by category:
   200246 kB: Dalvik
   172738 kB: Native
   151232 kB: Graphics
    89516 kB: Dalvik Other
    65741 kB: .dex mmap
    62904 kB: GL
    59426 kB: .so mmap
    58237 kB: Other dev
    24084 kB: Unknown
    15480 kB: .apk mmap
     8404 kB: Stack
     7562 kB: Other mmap
     1112 kB: Ashmem
     1099 kB: .ttf mmap
      160 kB: .jar mmap
       16 kB: Cursor
        0 kB: code mmap
        0 kB: image mmap
        0 kB: Memtrack

Total RAM: 1878788 kB
 Free RAM: 1258215 kB (402275 cached pss + 625628 cached + 230312 free)
 Used RAM: 780542 kB (515682 used pss + 192112 buffers + 3180 shmem + 69568 slab)
 Lost RAM: -159969 kB
   Tuning: 192 (large 512), oom 122880 kB, restore limit 40960 kB (high-end-gfx)
```

However, if you add a package name to the command (e.g., `adb shell dumpsys meminfo com.commonsware.books.android`), you will get a more detailed report about that specific app:

```
Applications Memory Usage (kB):
Uptime: 96120803 Realtime: 788242449

** MEMINFO in pid 12036 [com.commonsware.books.android] **
                   Pss  Private  Private  Swapped     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap     7642     7616        0        0     8732     8553      178
  Dalvik Heap     1920     1472        0        0     9860     9819       41
 Dalvik Other     1568     1428        0        0
        Stack      316      316        0        0
       Ashmem      128       68        0        0
    Other dev        4        0        4        0
     .so mmap     2372      528      116        0
    .apk mmap      130        0        8        0
    .ttf mmap       18        0        0        0
    .dex mmap     1232       12      936        0
   Other mmap      198        4        8        0
      Unknown     2516     2516        0        0
        TOTAL    18044    13960     1072        0    18592    18372      219
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
Objects
             Views:       48         ViewRootImpl:        1
       AppContexts:        3           Activities:        1
            Assets:        2        AssetManagers:        2
      Local Binders:        8        Proxy Binders:       17
   Death Recipients:        0
    OpenSSL Sockets:        0

SQL
        MEMORY_USED:       75
 PAGECACHE_OVERFLOW:        3          MALLOC_SIZE:       62

DATABASES
     pgsz     dbsz   Lookaside(b)          cache  Dbname
        1     6263             17         0/16/1  /data/data/
com.commonsware.books.android/databases/booksearch.db
```

This can show you:

- How much memory is being consumed by your Dalvik bytecode (`.dex mmap`), native libraries used directly by you or by framework components (`.so mmap`), etc.
- How many objects of various types are in the Dalvik heap, such as views and activities
- How much memory is used by SQLite for its page cache and related process-level buffers, plus which databases you have open that are contributing to memory consumption

Note that the combination of the `Private Dirty` and `Private Clean` columns is roughly analogous to the USS reported by `procstats`, in that it represents the amount of memory private to your process and that would be released should your process be terminated.

Measuring System RAM Consumption: Runtime

Some of the same information that the aforementioned reports contain is available at runtime via `ActivityManager` and other framework classes.

getMemoryInfo()

`getMemoryInfo()` on `ActivityManager` will fill in a supplied `ActivityManager.MemoryInfo` object. This will report to you:

- The total memory on the device (`totalMem`)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- The "available memory" (whose definition is a bit unclear) (`availMem`)
- What level of "available memory" is considered "low" and should trigger Android to start terminating processes beyond those that are cached, such as ones with running services (`threshold`)
- Whether we are presently in such a low-memory state (`lowMemory`)

getMyMemoryState()

`ActivityManager` also has a `getMyMemoryState()` method, on API Level 16+, that will populate an `ActivityManager.RunningAppProcessInfo` object with information about your process. While not everything in this object will be filled in, you will be able to get:

- The last trim level reported to your activities and `Application` via `onTrimMemory()` (`lastTrimLevel`)
- What importance level the OS considers your process to be in (`importance`), such as `IMPORTANCE_FOREGROUND` and `IMPORTANCE_EMPTY`
- For processes in the `IMPORTANCE_BACKGROUND` category — meaning the process has outstanding activities but is not in the foreground and has no service — the relative standing of the process compared to other background processes from a least-recently-used standpoint (`lru`), where lower numbers mean more recent usage

getProcessMemoryInfo()

The `getProcessMemoryInfo()` method on `ActivityManager` returns an array of `Debug.MemoryInfo` objects corresponding to the array of `int` process IDs (pids) that you pass in. The `Debug.MemoryInfo` objects report how much memory those identified processes are consuming. Of particular note, `getTotalPss()` returns the PSS for that process.

To get the `Debug.MemoryInfo` for your own process, you can use `getMemoryInfo()` on the `Debug` class, rather than find your own process ID and use `getProcessMemoryInfo()`. Or, on API Level 14+, you can simply call `getPss()` on `Debug` directly to find out your PSS.

Learn To Let Go (Of Your Heap)

Part of the reason for worrying a bit about your system RAM consumption is simply to "play nice" with the other apps that the user wants to use. However, since part of

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Android's decision-making about what processes to terminate tie into how much RAM those processes take up, the lower your system RAM footprint, the more likely it is that you can hang around for a while.

Part of reducing your system RAM consumption involves cleaning up your heap.

The reason the framework calls callback methods like `onTrimMemory()` is to help you reduce your heap usage to avoid `OutOfMemoryError` exceptions. However, allowing objects to be garbage-collected not only gives you more heap space, but it *also may reduce your system RAM footprint*.

To limit your system RAM usage, your process is not allocated all of its possible heap when the process is started up. Instead, the heap starts small and expands as you allocate more and more memory. However, the reverse is also true: if you release memory, the heap can shrink, returning RAM to the system. Android expands the heap on a "paged" basis, allocate more system RAM to add more pages to the heap. If, as a result of garbage collection, Dalvik sees that there are too many totally empty pages, Dalvik can free up those pages, returning them to the OS for use by other processes.

As noted in the chapter on the application heap, Dalvik's garbage collector is non-compacting, meaning that it does not move objects around to try to clean up pages or otherwise coalesce free memory blocks. Hence, a fragmented heap not only limits how well you can allocate new memory, but it also inhibits Dalvik's ability to reduce your system RAM usage.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Issues with Battery Life

Most Android devices are powered by batteries — Android TV is the biggest class of device that is not. Batteries are wonderful gizmos with one major problem: they are *always* running out of power.

Hence, users are very sensitive to battery consumption. Their ability to use their phones as actual *phones*, let alone for Android apps, depends on having enough battery power. The more apps drain the battery, the more frequently the user has to find a way to recharge the phone, and the more frequently the user fails and their phone shuts down.

The catch is that you may not notice the battery issues in your day-to-day development. The Android emulator's emulated battery does not drain based on you running your app. Your devices are often connected to your development machine via USB for testing and debugging, meaning they are perpetually being charged. Unless you are a regular user of your own app, you might not notice any increased power drain.

This part of the book is focused on helping you understand what is draining power and what you can do to be kinder and gentler on your users' batteries.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You're Getting Blamed

Users, for better or worse, have limited ability to determine what is responsible for draining the battery of their phone. Their #1 tool for this is the "Power Usage Summary" screen in the Settings app, sometimes referred to as the "battery blame screen".

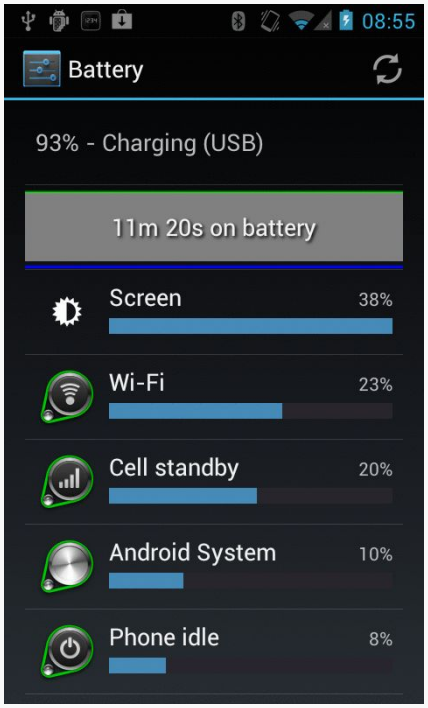


Figure 803: Battery Screen from Settings App

This lists both device features (e.g., the display) and applications. Android incrementally improves the accuracy of this screen with each passing release, trying to make sure the user understands what specifically is consuming the power.

If your application starts appearing on this screen, and the user does not feel that it is justified, the user is likely to become irritated with you.

Now, your appearance on this list might be perfectly reasonable. If you have written a video player app, and the user has just watched a few hours' worth of video, it is very likely that you will appear on this list and will be justified in your battery consumption.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, anything that you can do to not appear on this screen, or appear lower in the list, will help with user acceptance of your app.

This part of the book will show you how to measure your power usage and ways of trying to use less of it.

Not All Batteries Are Created Equal

Roughly speaking, battery capacity is proportional to screen size. Larger screens mean physically larger devices, and since the rest of the components (e.g., CPU) tend to be the same size, a larger device offers more room for a larger battery. This is good, as the screen is one of the major power draws on a device, and bigger screens draw more power.

Conversely, the battery on a "wearable" — whether eyewear like Google Glass, a smartwatch, or other form factors — tends to be much smaller than average, just because the wearables are physically smaller. A wearable is likely to have a battery with less than a third of the capacity of a phone, which in turn may have a battery with less than a third of the capacity of a large tablet.

Hence, depending upon where your app will be running, the amount of battery available in total will vary widely. What might be considered acceptable battery consumption on a tablet would be considered excessive on a wearable.

Stretching Out the Last mWh

Sometimes, what the user wants your app to do in one case is not what the user wants your app to do in other cases. Serious power-draining might be reserved for when the device is plugged in, or when the device has at least such-and-so power remaining. The user may value the last milliwatt-hours (mWh) more than others and want your application to use less power in those circumstances.

Hence, if your application polls the Internet, you might offer a feature to poll less frequently, or perhaps not at all, when power is low. If your application uses GPS to find a location (e.g., automatic "check-ins" to social networks like Foursquare), you might offer to skip such actions when the battery is low. You might want to signal to the user when the battery gets low during playback of a video, or during the game they are in. And so on.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This part of the book will help you identify when the battery is low and strategies for making use of that information.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Power Measurement Options

As with any situation where you are trying to reduce your use of some system resource, you need to be able to accurately measure how much you are using that resource. Otherwise, you will have no idea whether your attempts to reduce usage are helping. It is possible that what you think will consume less of the resource actually consumes *more*, because of unanticipated side-effects. And, if nothing else, if the change makes your code more complicated and does not help much with resource consumption, you may be better served sticking with the original, simpler implementation.

So, when it comes to power usage, it helps to know how much power you are consuming, to determine if your attempts to use less power actually do help.

Unfortunately, compared to things like RAM and bandwidth, power measurement is a significant challenge. You really need to have hardware specifically instrumented to report power consumption for pieces of that hardware (CPU versus screen versus GPS versus mobile data radio versus ...). Even if you cannot get power usage per component, just having accurate power consumption overall is not something you can necessarily get from any Android device. Alas, getting that level of power usage knowledge can be troublesome in its own right, for a variety of reasons.

This chapter will explore a few ways of measuring power usage, along with the pros and cons of that approach.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

batterystats and the Battery Historian

Android 5.0 brought us "Project Volta", an initiative to reduce the amount of power consumed by apps, the framework classes, and the OS itself.

Part of what we got from Project Volta is `batterystats`, a dump of data pertaining to power consumption. Since that data dump can be large and inscrutable, we also have the Battery Historian, a Python script that can convert key `batterystats` output into a timeline of events.

However, none of this is especially well-documented at this time, and so the usefulness of these utilities is limited at present. The following sections provide some basic guidance for trying to use these tools.

Running a Test

First, since `batterystats` is obtained via `adb shell dumpsys`, you will want `adb` to be in your `PATH`, by adding your SDK installation's `platform-tools/` directory to your `PATH` environment variable.

Then, in a terminal, run:

```
adb shell dumpsys batterystats --enable full-wake-history
adb shell dumpsys batterystats --reset
```

This will ensure that `batterystats` captures all relevant information about `WakeLock` behavior, and it resets all of the logs.

At this point, run your tests. Ideally, you would do so in a fairly power-neutral environment, such as not using a micro USB cable for an `adb` connection (as that charges the device).

When your test scenario is complete, run `adb shell dumpsys batterystats`, redirecting the output to some file:

```
adb shell dumpsys batterystats > /tmp/bs.txt
```

You can optionally supply your `applicationId` as part of the `batterystats` command, which will restrict the output to events pertaining to your app. However, some events that are from other processes, like the Play Services Framework, may be

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

of interest to you. You will need to experiment to determine which mode (full or filtered for your app) will work best for you.

Interpreting the Text Output

Depending on how long your test runs, the information included in the `batterystats` output can be anywhere from tens of KB to tens of MB in size.

Battery History

The file will lead off with the "Battery History" section:

```
Battery History (13% used, 35KB used of 256KB, 68 strings using 4232):
          0 (9) RESET:TIME: 2014-10-25-19-54-07
          0 (2) 100 status=not-charging health=good plug=none temp=207
volt=4296 +running +wake_lock +sensor +phone_scanning +audio +screen
phone_state=out +wifi_running +wifi wifi_signal_strength=4
wifi_suppl=completed proc=u0a3:"android.process.acore"
          0 (2) 100 proc=u0a29:"com.android.calendar"
          0 (2) 100 proc=1027:"com.android.nfc:sendui"
          0 (2) 100 proc=u0a7:"com.google.android.gms"
          0 (2) 100 proc=1000:"WebViewLoader-armeabi-v7a"
          0 (2) 100 proc=u0a32:"com.google.android.configupdater"
          0 (2) 100 proc=u0a7:"com.google.process.location"
          0 (2) 100 proc=u0a12:"com.android.launcher"
          0 (2) 100 proc=1001:"com.android.server.telecom"
          0 (2) 100 proc=u0a7:"com.google.process.gapps"
          0 (2) 100 proc=u0a55:"com.nuance.xt9.input"
          0 (2) 100 proc=u0a33:"com.google.android.deskclock"
          0 (2) 100 proc=u0a5:"android.process.media"
          0 (2) 100 proc=u0a20:"com.android.systemui"
          0 (2) 100 proc=1027:"com.android.nfc"
          0 (2) 100 proc=1001:"com.android.phone"
          0 (2) 100 proc=u0a37:"com.google.android.gallery3d"
          0 (2) 100 proc=u0a58:"com.commonsware.android.job"
          0 (2) 100 proc=u0a2:"com.android.providers.calendar"
          0 (2) 100 proc=u0a16:"com.android.vending"
          0 (2) 100 proc=u0a7:"com.google.android.gms.unstable"
          0 (2) 100 proc=u0a42:"com.google.android.inputmethod.latin"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
          0 (2) 100 proc=u0a11:"com.google.android.partnersetup"
          0 (2) 100 top=u0a58:"com.commonsware.android.job"
          0 (2) 100 wake_lock_in=-1:"screen"
          0 (2) 100 user=0:"0"
          0 (2) 100 userfg=0:"0"
      +36ms (2) 100 +wake_lock_in=u0a7:"Wakeful StateMachine:
GeofencerStateMachine"
      +37ms (2) 100 -wake_lock_in=u0a7:"Wakeful StateMachine:
GeofencerStateMachine"
     +104ms (2) 100 +wake_lock_in=u0a7:"UlrDispatchingService"
     +146ms (2) 100 +wake_lock_in=u0a7:"GCoreFlp"
     +147ms (2) 100 -wake_lock_in=u0a7:"GCoreFlp"
     +150ms (2) 100 -wake_lock_in=u0a7:"UlrDispatchingService"
   +2s001ms (2) 100 volt=4243
   +6s779ms (3) 100 -sensor +wake_lock_in=1000:"ActivityManager-Sleep"
   +6s781ms (2) 100 +wake_lock_in=u0a20:"show keyguard"
   +6s811ms (2) 100 -wake_lock_in=1000:"ActivityManager-Sleep"
   +6s820ms (2) 100 +wake_lock_in=1000:"WifiSuspend"
   +6s835ms (2) 100 -wake_lock_in=1000:"WifiSuspend"
   +6s868ms (2) 100 +wake_lock_in=1013:"AudioMix"
   +6s875ms (2) 100 -wake_lock_in=u0a20:"show keyguard"
   +6s878ms (2) 100 -wake_lock_in=1013:"AudioMix"
   +6s878ms (2) 100 +wake_lock_in=u0a20:"AudioMix"
   +6s941ms (2) 100 +wake_lock_in=1027:"NfcService:mRoutingWakeLock"
   +6s941ms (2) 100 +wake_lock_in=u0a7:"Wakeful StateMachine:
GeofencerStateMachine"
   +6s942ms (2) 100 -wake_lock_in=u0a7:"Wakeful StateMachine:
GeofencerStateMachine"
   +6s943ms (2) 100 +wake_lock_in=u0a7:"GCoreFlp"
   +6s946ms (2) 100 -wake_lock_in=u0a7:"GCoreFlp"
   +6s968ms (2) 100 -wake_lock_in=1027:"NfcService:mRoutingWakeLock"
.
.
.
```

This contains information about how much the battery was drained during the test run, along with a detailed roster of the power-related events that occurred during the test run. The timestamps on those roster entries are relative to the first entry in the roster. Beyond that, there is little explanation of what the roster entries mean.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Per-PID Stats

Next, there will be a short stanza labeled "Per-PID Stats" and "Discharge step durations":

```
Per-PID Stats:
  PID 0 wake time: +134ms
  PID 536 wake time: +1m35s993ms
  PID 0 wake time: +10s842ms
  PID 881 wake time: +301ms
  PID 536 wake time: +70ms
  PID 989 wake time: +23s167ms
  PID 1136 wake time: +2s974ms
  PID 1193 wake time: +230ms
  PID 0 wake time: +1s123ms
  PID 617 wake time: +187ms
  PID 536 wake time: +18ms
  PID 536 wake time: +13ms
  PID 627 wake time: +586ms
  PID 536 wake time: +184ms
  PID 3690 wake time: +9m42s965ms

Discharge step durations:
  #0: +4h8m36s976ms to 97 (screen-off, power-save-off)
  #1: +3h7m47s132ms to 98 (screen-off, power-save-off)
```

If you determine your process' PID, you will see how long the process' "wake time" was. The precise definition of "wake time" is undocumented.

"Statistics since last charge" Summary

Next up will be a "Statistics since last charge" header, with a few summary blocks of data:

```
Statistics since last charge:
  System starts: 0, currently on battery: false
  Time on battery: 11h 30m 29s 177ms (99.9%) realtime, 16m 13s 306ms (2.3%)
uptime
  Time on battery screen off: 11h 30m 7s 940ms (99.9%) realtime, 15m 52s 69ms
(2.3%) uptime
  Total run time: 11h 30m 55s 300ms realtime, 16m 39s 430ms uptime
  Start clock time: 2014-10-25-19-54-07
  Screen on: 21s 237ms (0.1%) 2x, Interactive: 20s 191ms (0.0%)
  Screen brightnesses:
    dark 21s 237ms (100.0%)
  Total partial wakelock time: 10m 22s 877ms
  Mobile total received: 0B, sent: 0B (packets received 0, sent 0)
  Phone signal levels:
    none 11h 30m 29s 177ms (100.0%) 0x
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
Signal scanning time: 9s 0ms
Radio types:
  none 11h 30m 29s 177ms (100.0%) 0x
Mobile radio active time: 0ms (0.0%) 0x
Wi-Fi total received: 0B, sent: 0B (packets received 0, sent 0)
Wifi on: 11h 30m 29s 177ms (100.0%), Wifi running: 11h 30m 29s 177ms (100.0%)
Wifi states: (no activity)
Wifi supplicant states:
  group-handshake 16ms (0.0%) 12x
  completed 11h 30m 29s 161ms (100.0%) 12x
Wifi signal levels:
  level(4) 11h 30m 29s 177ms (100.0%) 1x
Bluetooth on: 0ms (0.0%)
Bluetooth states: (no activity)

Device battery use since last full charge
  Amount discharged (lower bound): 2
  Amount discharged (upper bound): 3
  Amount discharged while screen on: 0
  Amount discharged while screen off: 3

Estimated power use (mAh):
  Capacity: 3448, Computed drain: 107, actual drain: 69.0-103
  Idle: 40.3
  Wifi: 36.4
  Uid u0a58: 14.8
  Uid 0: 12.8
  Uid 1000: 1.75
  Uid u0a20: 0.521
  Uid u0a7: 0.389
  Screen: 0.375
  Uid 1013: 0.0775
  Uid 1001: 0.0219
  Uid u0a42: 0.0168
  Uid u0a12: 0.0138
  Uid 1027: 0.0110
  Uid u0a5: 0.00866
  Uid u0a33: 0.00253
  Uid u0a16: 0.000867
  Uid u0a3: 0.000815
  Uid u0a29: 0.000523
  Uid u0a2: 0.000474
  Over-counted: 4.04
```

The first block has useful data about how much various radios were on, how much data they transmitted, how long the screen was on, how long the device had an outstanding partial `WakeLock`, etc.

Also, the “Estimated power use (mAh)” block is basically the data that underlies the “battery blame screen” in Settings. You will see how many milliamp-hours (mAh) were attributed to your process.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

WakeLock Summary

Next up is a summary of `WakeLock` events:

```
All kernel wake locks:
Kernel Wake lock PowerManagerService.WakeLocks: 10m 23s 46ms (717 times) realtime
Kernel Wake lock qcom_rx_wakelock: 9m 9s 836ms (1537 times) realtime
Kernel Wake lock alarm_rtc: 55s 327ms (717 times) realtime
Kernel Wake lock sns_async_ev_wakelock: 16s 525ms (9 times) realtime
Kernel Wake lock power-supply: 15s 730ms (1448 times) realtime
Kernel Wake lock event0-536: 11s 822ms (1484 times) realtime
Kernel Wake lock event2-536: 10s 686ms (1509 times) realtime
Kernel Wake lock event4-536: 10s 234ms (1509 times) realtime
Kernel Wake lock alarm: 7s 972ms (1239 times) realtime
Kernel Wake lock wlan: 1s 608ms (2 times) realtime
Kernel Wake lock PowerManagerService.Display: 611ms (2 times) realtime
Kernel Wake lock main: 608ms (0 times) realtime
Kernel Wake lock KeyEvents: 136ms (1548 times) realtime
Kernel Wake lock mmc0_detect: 38ms (1507 times) realtime
Kernel Wake lock deleted_wake_locks: 28ms (170 times) realtime
Kernel Wake lock event5-536: 20ms (2 times) realtime

All partial wake locks:
Wake lock u0a58 wake:com.commonsware.android.job/.DemoScheduledService: 9m 0s
153ms (671 times) realtime
Wake lock 1000 *alarm*: 44s 114ms (683 times) realtime
Wake lock u0a7 Checkin Service: 16s 572ms (4 times) realtime
Wake lock 1000 NetworkStats: 7s 347ms (338 times) realtime
Wake lock 1013 AudioMix: 5s 833ms (2 times) realtime
Wake lock 1000 DHCP: 4s 953ms (12 times) realtime
Wake lock u0a7 *net_scheduler*: 1s 504ms (82 times) realtime
Wake lock u0a7 Event Log Service: 1s 98ms (18 times) realtime
Wake lock u0a42 DownloadManager: 322ms (1 times) realtime
Wake lock u0a7 Config Service fetch: 235ms (1 times) realtime
Wake lock u0a7 Icing: 197ms (5 times) realtime
Wake lock u0a7 Event Log Handoff: 143ms (18 times) realtime
Wake lock u0a58 *alarm*: 120ms (24 times) realtime
Wake lock u0a7 GCM_CONN: 61ms (36 times) realtime
Wake lock u0a7 GmsDownloadService: 47ms (1 times) realtime
Wake lock u0a7 *alarm*: 42ms (10 times) realtime
Wake lock u0a7 Wakeful StateMachine: GeofencerStateMachine: 35ms (8 times)
realtime
Wake lock 1000 SyncManagerHandleSyncAlarm: 34ms (6 times) realtime
Wake lock u0a7 GCM_HB_ALARM: 34ms (36 times) realtime
Wake lock u0a7 Checkin Handoff: 12ms (4 times) realtime
Wake lock 1000 SyncLoopWakeLock: 8ms (4 times) realtime
Wake lock u0a33 *alarm*: 7ms (4 times) realtime
Wake lock u0a42 *alarm*: 4ms (3 times) realtime
Wake lock u0a7 GCoreFlp: 2ms (5 times) realtime
```

Your code will tend to show up in the "All partial wake locks" section, showing how many `WakeLocks` you acquired and for how long overall.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

And, if you show up here, you can definitely find out your app's PID — for example, `u0a58` is associated with the `com.commonsware.android.job` package.

Per PID Summary

Next up are summaries of information per process. Your process will show up somewhere in the list:

```
.
.
.
u0a58:
  Wake lock wake:com.commonsware.android.job/.DemoScheduledService: 9m 0s 153ms
partial (671 times) realtime
  Wake lock *alarm*: 120ms partial (24 times) realtime
  TOTAL wake: 9m 0s 273ms partial realtime
  Foreground activities: 7s 817ms realtime (1 times)
  Foreground for: 12s 959ms
  Active for: 3h 58m 47s 853ms
  Running for: 11h 30m 29s 177ms
  Proc com.commonsware.android.job:
    CPU: 1m 4s 370ms usr + 21s 140ms krn ; 400ms fg
  Proc *wakelock*:
    CPU: 43s 690ms usr + 1m 19s 590ms krn ; 0ms fg
  Apk com.commonsware.android.job:
    672 wakeup alarms
    Service com.commonsware.android.job.DemoScheduledService:
      Created for: 11m 14s 174ms uptime
      Starts: 647, launches: 647
.
.
.
```

As usual, the exact definitions of the information here is largely undocumented.

Installing and Running the Battery Historian

While `batterystats` is part of the Android 5.0 runtime environment, and tools like `adb` are part of the Android SDK, the Battery Historian is neither. Instead, it is a Python script that you have to download to your development machine from its GitHub project.

Given that, you can run the Python script, supplying it with the output of your `batterystats` run, and redirecting the script's output to an HTML file:

```
python historian.py /tmp/bs.txt > /tmp/bs-report.html
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Interpreting the Historian Output

You can then load that HTML into a Web browser (Chrome-flavored ones are probably a good choice, given that it is Google-generated HTML). This will give you a timeline across the horizontal axis, with event categories culled from the "Battery History" section of the `batterystats` output on the vertical axis:

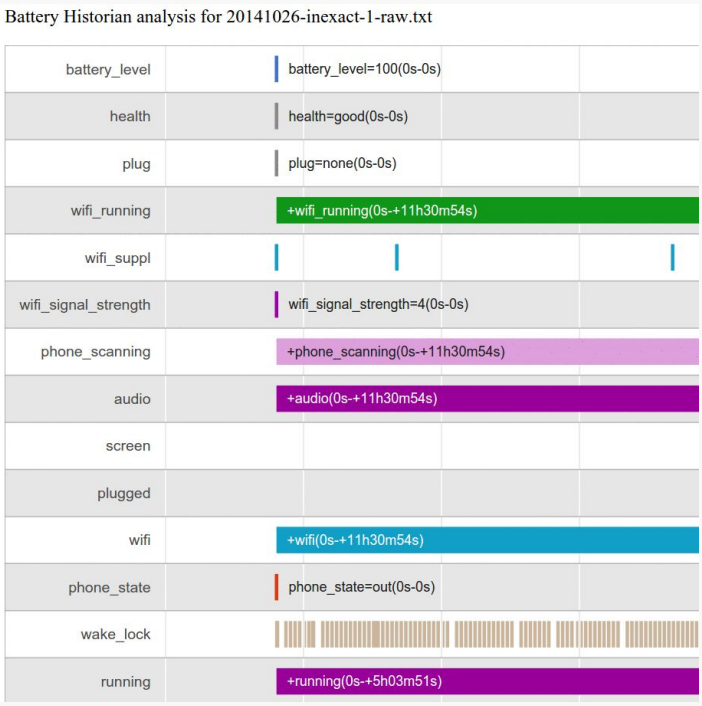


Figure 804: Battery Historian Timeline, Partial View

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 805: Battery Historian Timeline, Additional

The length of the bar shows the approximate duration of the event, though really short events have a *de minimus* length to give you something to see. Hovering your mouse over one of the bars brings up a pop-up with more details about that event:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

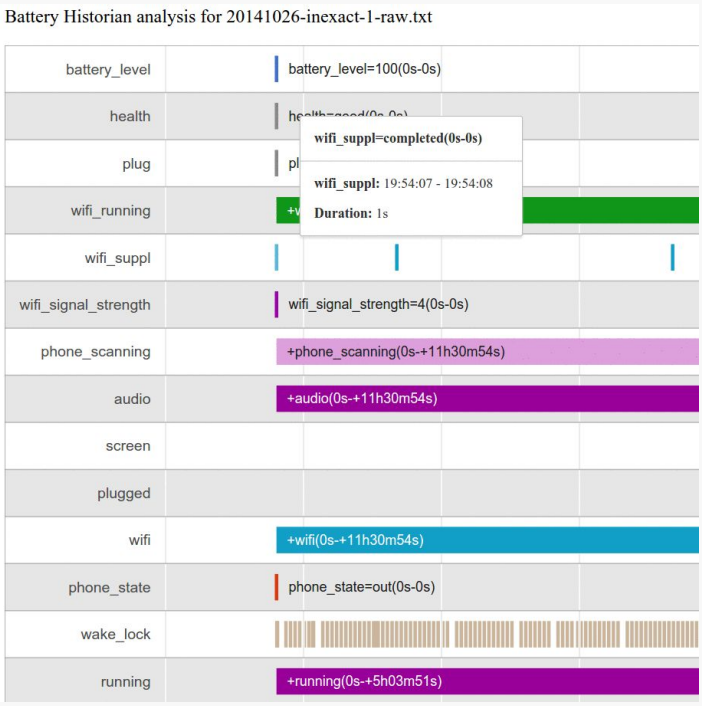
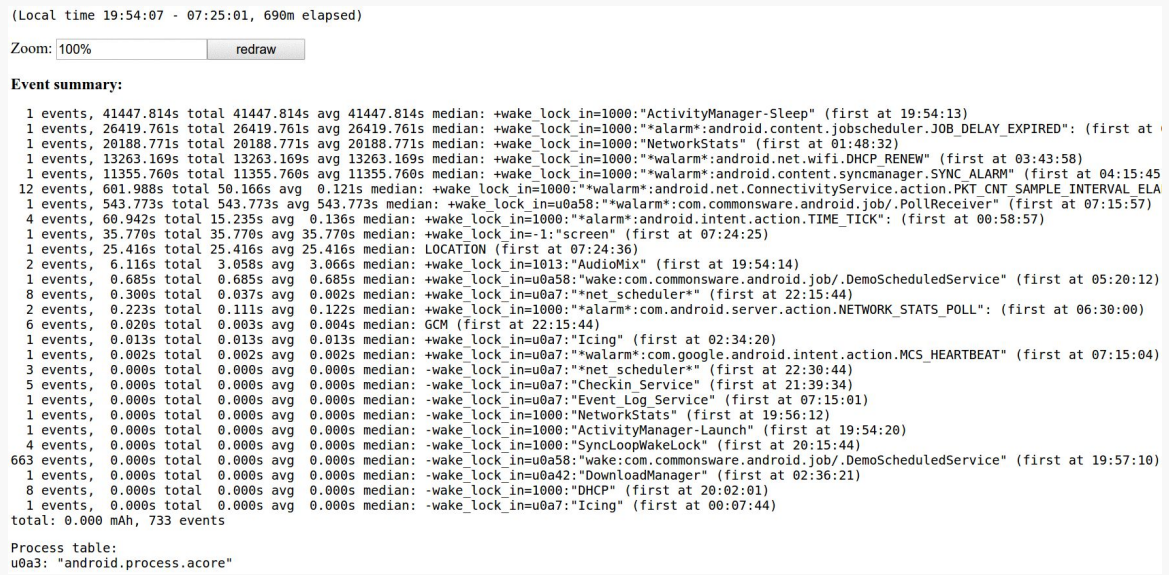


Figure 806: Battery Historian Timeline, Partial View, with Pop-Up

If the bar is really long, you may need to scroll your browser horizontally to see the pop-up, as the rendering of the pop-up location does not seem to pay attention to the browser viewport very well.

Below the main chart is a “Zoom” field that you can use to change the scale of the horizontal axis, along with an “Event summary”:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition



(Local time 19:54:07 - 07:25:01, 690m elapsed)

Zoom: 100% redraw

Event summary:

```
  1 events, 41447.814s total 41447.814s avg 41447.814s median: +wake_lock_in=1000:"ActivityManager-Sleep" (first at 19:54:13)
  1 events, 26419.761s total 26419.761s avg 26419.761s median: +wake_lock_in=1000:"*alarm*:android.content.jobscheduler.JOB_DELAY_EXPIRED": (first at
  1 events, 20188.771s total 20188.771s avg 20188.771s median: +wake_lock_in=1000:"NetworkStats" (first at 01:48:32)
  1 events, 13263.169s total 13263.169s avg 13263.169s median: +wake_lock_in=1000:"*walarm*:android.net.wifi.DHCP_RENEW" (first at 03:43:58)
  1 events, 11355.760s total 11355.760s avg 11355.760s median: +wake_lock_in=1000:"*walarm*:android.content.syncmanager.SYNC_ALARM" (first at 04:15:45
 12 events, 601.988s total 50.166s avg  0.121s median: +wake_lock_in=1000:"*walarm*:android.net.ConnectivityService.action.PKT_CNT_SAMPLE_INTERVAL_ELA
  1 events, 543.773s total 543.773s avg 543.773s median: +wake_lock_in=u0a58:"*walarm*:com.commonsware.android.job/.PollReceiver" (first at 07:15:57)
  4 events, 60.942s total 15.235s avg  0.136s median: +wake_lock_in=1000:"*alarm*:android.intent.action.TIME_TICK": (first at 00:58:57)
  1 events, 35.770s total 35.770s avg 35.770s median: +wake_lock_in=-1:"screen" (first at 07:24:25)
  1 events, 25.416s total 25.416s avg 25.416s median: LOCATION (first at 07:24:36)
  2 events,  6.116s total  3.058s avg  3.066s median: +wake_lock_in=1013:"AudioMix" (first at 19:54:14)
  1 events,  0.685s total  0.685s avg  0.685s median: +wake_lock_in=u0a58:"wake:com.commonsware.android.job/.DemoScheduledService" (first at 05:20:12)
  8 events,  0.300s total  0.037s avg  0.002s median: +wake_lock_in=u0a7:"*net_scheduler*" (first at 22:15:44)
  2 events,  0.223s total  0.111s avg  0.122s median: +wake_lock_in=1000:"*alarm*:com.android.server.action.NETWORK_STATS_POLL": (first at 06:30:00)
  6 events,  0.020s total  0.003s avg  0.004s median: GCM (first at 22:15:44)
  1 events,  0.013s total  0.013s avg  0.013s median: +wake_lock_in=u0a7:"Icing" (first at 02:34:20)
  1 events,  0.002s total  0.002s avg  0.002s median: +wake_lock_in=u0a7:"*walarm*:com.google.android.intent.action.MCS_HEARTBEAT" (first at 07:15:04)
  3 events,  0.000s total  0.000s avg  0.000s median: -wake_lock_in=u0a7:"*net_scheduler*" (first at 22:30:44)
  5 events,  0.000s total  0.000s avg  0.000s median: -wake_lock_in=u0a7:"Checkin_Service" (first at 21:39:34)
  1 events,  0.000s total  0.000s avg  0.000s median: -wake_lock_in=u0a7:"Event_Log_Service" (first at 07:15:01)
  1 events,  0.000s total  0.000s avg  0.000s median: -wake_lock_in=1000:"NetworkStats" (first at 19:56:12)
  1 events,  0.000s total  0.000s avg  0.000s median: -wake_lock_in=1000:"ActivityManager-Launch" (first at 19:54:20)
  4 events,  0.000s total  0.000s avg  0.000s median: -wake_lock_in=1000:"SyncLoopWakeLock" (first at 20:15:44)
663 events,  0.000s total  0.000s avg  0.000s median: -wake_lock_in=u0a58:"wake:com.commonsware.android.job/.DemoScheduledService" (first at 19:57:10)
  1 events,  0.000s total  0.000s avg  0.000s median: -wake_lock_in=u0a42:"DownloadManager" (first at 02:36:21)
  8 events,  0.000s total  0.000s avg  0.000s median: -wake_lock_in=1000:"DHCP" (first at 20:02:01)
  1 events,  0.000s total  0.000s avg  0.000s median: -wake_lock_in=u0a7:"Icing" (first at 00:07:44)
total: 0.000 mAh, 733 events

Process table:
u0a3: "android.process.acore"
```

Figure 807: Battery Historian Timeline, Zoom and "Event summary"

Again, this is largely undocumented.

The Qualcomm Tool (That Must Not Be Named)

Qualcomm makes the chipsets that drive a substantial percentage of Android devices. Qualcomm also offers a number of things to further help Android developers, such as open source libraries like AllJoyn (P2P communications).

Qualcomm makes a tool that can help you measure power consumption. There are two versions of this tool:

1. The best version is designed to run on the Qualcomm MDP series of reference hardware, and takes advantage of specific capabilities of that hardware to provide excellent power consumption data
2. Another version can be used with other Android devices powered by Qualcomm chipsets, providing information about power usage, though not quite as much as you get from the first version

Alas, this book does not cover this tool.

The current versions of this tool have a license agreement containing crudely-implemented non-disclosure terms:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

> You shall not to disclose or permit the disclosure of the Materials in any form or any information relating to the Materials (including without limitation the results of use or testing) to any third party without QTI's prior written permission... You further acknowledge and agree that if QTI, in its sole discretion, chooses to provide any form of support or information relating to the Materials, such support and information shall be deemed confidential and proprietary to QTI and shall be protected in accordance with this Section 4.

In this case, "Materials" refers to the two versions of the tool, plus accompanying documentation, and QTI is Qualcomm Technologies.

The non-disclosure terms do not come with the normal "you can talk about anything that has been already publicly disclosed" caveats. Hence, even the Qualcomm public Web site regarding this tool falls under the "any form of support or information".

Regardless, you may be interested in visiting Qualcomm's developer support site in hopes that you can find this unnamed tool.

PowerTutor

Perhaps the best-known third-party power analyzer is PowerTutor. PowerTutor is the outcome of a research project from the University of Michigan, with a bit of assistance from Google. In principle, PowerTutor is capable of letting you know power consumption on a device, much along the lines of what Trepn can record on a Qualcomm MDP. In practice, PowerTutor is significantly less powerful and sophisticated.

PowerTutor was created with the HTC Dream (T-Mobile G1), HTC Magic (T-Mobile G2), and Nexus One in mind. Its power output values will be as accurate as they could make it for those devices. If you run PowerTutor on other hardware, the results will be less accurate.

You can obtain PowerTutor from the Play Store, or from the PowerTutor Web site, or you can compile it from source.

PowerTutor is not tied to testing a particular application. As such, you can simply run PowerTutor whenever you want from its launcher icon, then press "Start Power Profiler" in the main activity:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

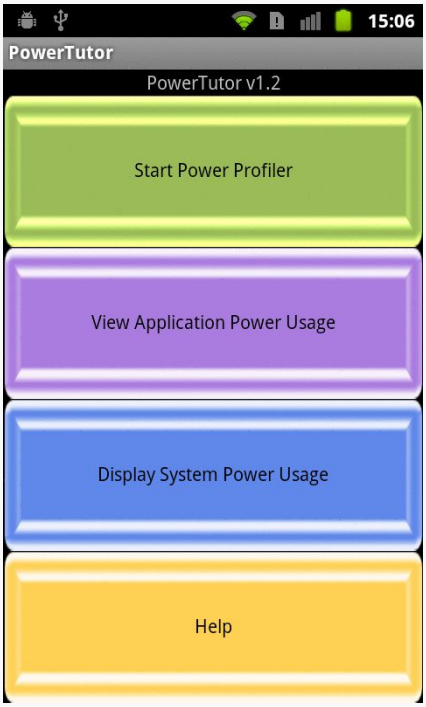


Figure 808: The PowerTutor main activity

At this point, you can start playing with your application, or running your unit test suite, or whatever. When you want to get an idea of how much power you have been consuming, you can switch back to the PowerTutor activity and choose "View Application Power Usage". This brings up a list of processes and toggle buttons to show various power consumption values for each:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

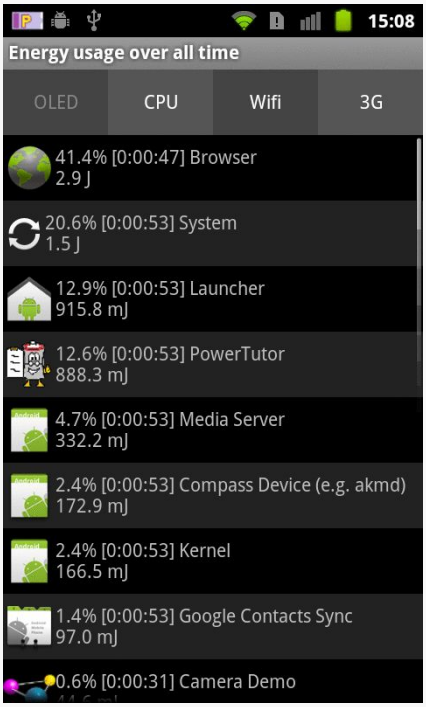


Figure 809: The PowerTutor application roster

Tapping the list entry brings up a graph for that particular process, though since this information is only available while PowerTutor is recording new data, the graph is usually empty unless you have logic running in the background:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

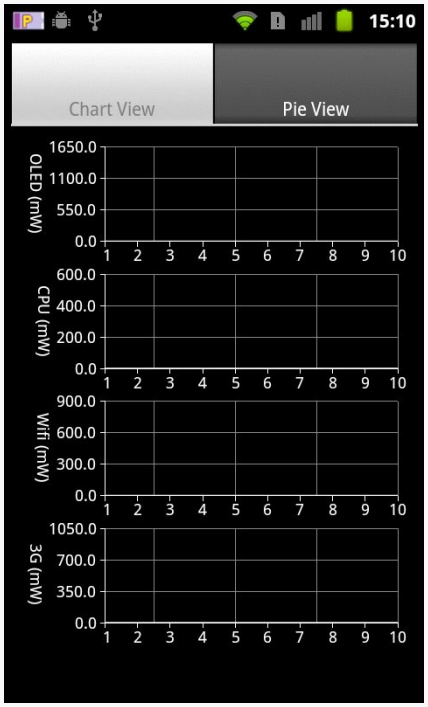


Figure 810: The PowerTutor live charts for a single process current power consumption

You can also bring up a chart showing what portion of your power consumption came from various sources for the whole device, such as a pie chart of current consumption:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

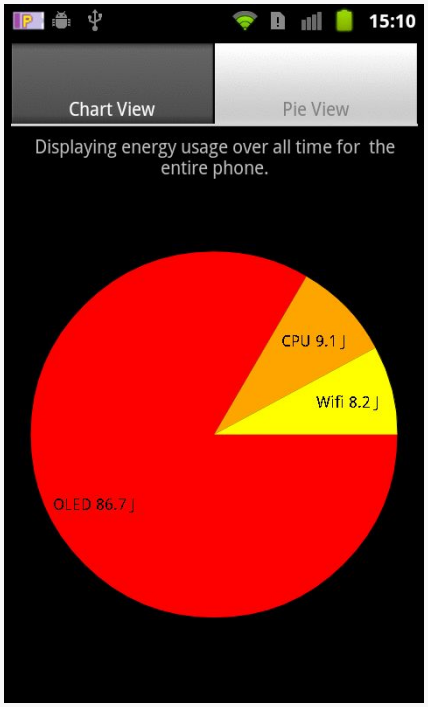


Figure 811: The PowerTutor pie chart for current overall power consumption

Given that the source code is available, one might augment PowerTutor to:

1. Saving results, both as data files for offline analysis (akin to Trepn's CSV files) or for viewing charts and tables on the device when data is not being actively collected
2. Allowing one to record application states, akin to Trepn, to better correlate application functionality to saved power results

Battery Screen in Settings Application

Of course, what developers tend to focus on most with power is the battery consumption screen in the Settings application, as shown in a previous chapter:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

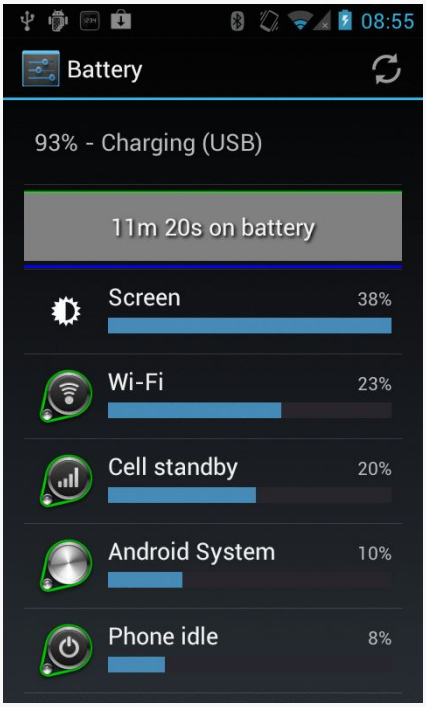


Figure 812: Battery Screen from Settings App

After all, this is what users will tend to focus on — anything showing up in here is a source of blame for whatever power woes the user believes she is experiencing. Conversely, if your application does not show up in this screen during normal operation, then there is no compelling reason for you to do further analysis, as users will tend to be oblivious to your actual power consumption.

If you do show up in the list, tapping on your entry can give you some more details of what power you consumed and why:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

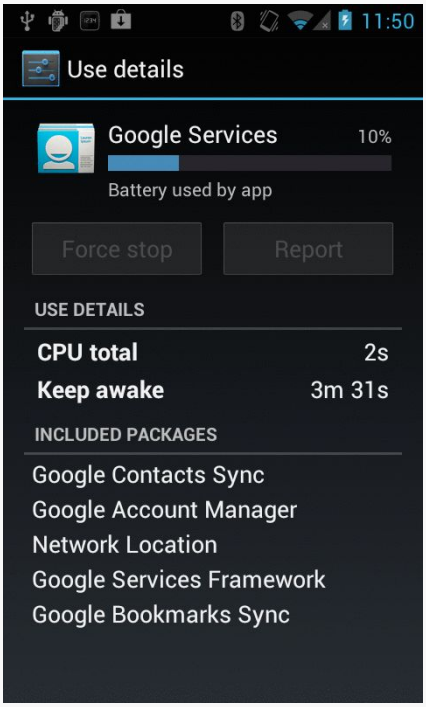


Figure 813: Battery Details Screen from Settings App

However, the information contained in here is mostly guesswork, using a more refined version of the same approach that PowerTutor uses. Ordinary Android hardware simply lacks enough fine-grained power measurement instrumentation to do an accurate job of apportioning power usage among different processes. So, the details of how long you kept the CPU powered on may be accurate, but the percentage of battery consumption associated with your app is just an estimate.

BatteryInfo Dump

Yet another possibility is to use the `adb shell dumpsys batteryinfo` command from your command prompt or terminal on your development workstation. This will emit a fair amount of data that probably means something to somebody, such as general device information:

```
Battery History:
       -1h00m56s463ms 096 20030002 status=discharging health=good
plug=none temp=191 volt=4060 +screen +wake_lock +sensor
brightness=medium
       -1h00m52s490ms 096 22030302 +wifi phone_state=off
       -1h00m51s844ms 096 2703d102 +phone_scanning +wifi_running
phone_state=out data_conn=other
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
       -1h00m49s303ms 096 2743d102 +wifi_scan_lock
         -57m48s766ms 095 2743d102
         -53m24s627ms 095 2743d100 brightness=dark
         -53m17s620ms 095 0741d100 -screen -wake_lock
         -53m17s107ms 095 0740d100 -sensor
         -38m17s007ms 095 0642d100 -wifi_running +wake_lock
         -38m08s998ms 095 0640d100 -wake_lock
            -54s781ms 095 4640d100 status=full plug=usb temp=193
volt=4084 +plugged

Per-PID Stats:
  PID 96 wake time: +12s75ms
  PID 177 wake time: +1s13ms
  PID 458 wake time: +1s898ms
  PID 326 wake time: +3s925ms
  PID 205 wake time: +2s107ms
  PID 415 wake time: +843ms
  PID 96 wake time: +281ms

Statistics since last charge:
  System starts: 0, currently on battery: false
  Time on battery: 1h 0m 1s 682ms (0.3%) realtime, 8m 21s 883ms
(0.0%) uptime
  Total run time: 16d 11h 13m 34s 654ms realtime, 2h 9m 37s 404ms
uptime,
  Screen on: 7m 37s 868ms (12.7%), Input events: 0, Active phone
call: 0ms (0.0%)
  Screen brightnesses: dark 7s 7ms (1.5%), medium 7m 30s 861ms (98.5%)
  Kernel Wake lock "SMD_DS": 2s 368ms  (3 times) realtime
  Kernel Wake lock "mmc_delayed_work": 1s 210ms  (1 times) realtime
  Kernel Wake lock "SMD_RPCCALL": 56ms  (435 times) realtime
  Kernel Wake lock "power-supply": 575ms  (4 times) realtime
  Kernel Wake lock "radio-interface": 3s 1ms  (3 times) realtime
  Kernel Wake lock "ApmCommandThread": 4ms  (10 times) realtime
  Kernel Wake lock "ds2784-battery": 2s 6ms  (21 times) realtime
  Kernel Wake lock "msmfb_idle_lock": 14ms  (2273 times) realtime
  Kernel Wake lock "kgsl": 51s 482ms  (613 times) realtime
  Kernel Wake lock "rpc_read": 164ms  (272 times) realtime
  Kernel Wake lock "main": 7m 39s 708ms  (0 times) realtime
  Total received: 0B, Total sent: 0B
  Total full wakelock time: 149ms , Total partial waklock time: 31s
14ms
  Signal levels: none 59m 57s 63ms (99.9%) 1x
  Signal scanning time: 59m 57s 63ms
  Radio types: none 641ms (0.0%) 1x, other 59m 56s 973ms (99.9%) 1x
  Radio data uptime when unplugged: 0 ms
  Wifi on: 59m 57s 709ms (99.9%), Wifi running: 22m 35s 424ms
(37.6%), Bluetooth on: 0ms (0.0%)

  Device battery use since last full charge
    Amount discharged (lower bound): 0
    Amount discharged (upper bound): 1
    Amount discharged while screen on: 1
    Amount discharged while screen off: 0
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
(... and lots more...)
```

and per-process information (here, showing power used by PowerTutor itself):

```
#10058:
    Wake lock window: 5s 71ms window (1 times) realtime
    Proc edu.umich.PowerTutor:
      CPU: 11s 750ms usr + 4s 530ms krn
      1 proc starts
    Apk edu.umich.PowerTutor:
      Service edu.umich.PowerTutor.service.UMLoggerService:
        Created for: 4m 4s 750ms  uptime
        Starts: 1, launches: 1
```

In principle, one might create tools that use this output — or perhaps steal a peek at the data used by the Settings application – to create something a bit more developer-friendly.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Sources of Power Drain

If you can measure power drain well yourself, that is the best way for you to determine precisely where your power consumption is going. Alas, for various reasons, you may not be able to get good power consumption data.

Which means you may have to guess.

We know the general sorts of things that consume power in a device, such as the screen and the CPU. We know that if we use these things less, we will use less power. Eventually, though, we have an app that does nothing, and while this may result in optimal power usage, we are still likely to get poor reviews, because *the app does nothing*.

What we need is some rough idea of how bad certain things are, so we can weigh our use of those system components appropriately.

This chapter will try to give you some "rule of thumb" heuristics of how to estimate power usage of various system components, plus some general recommendations of how to use less of that particular component without necessarily eliminating useful functionality from your app.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

Also note that:

- mA = milliamps, where the ampere (or "amp") is the SI unit of current

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- mAH = milliamp-hours, which is how battery capacities are measured (e.g., 2000mAH can power a 200mA draw for 10 hours)

Screen

Screen size and battery size generally trend together. Tablets have bigger batteries and bigger screens than do phones, which in turn are bigger in both areas than are wearables.

A rough rule of thumb is to expect to consume ~10% of the device's battery for every hour you keep the screen on. Or, to look at it another way, on a phone-sized screen, expect a power draw of ~100–200mA, depending on variations in screen size and display technology (e.g., AMOLED).

Normally, the user is in control over how long your app is in the foreground and therefore is "to blame" for the screen being on. There are a couple of cases where you can make the screen be more of a problem.

The first is if you `acquire()` a `WakeLock` (other than a `PARTIAL_WAKE_LOCK`)... and forget to ever `release()` it. Since the `WakeLock` will keep the screen on, the screen will stay on, even if your app is in the background, until such time as your process is terminated or the device shuts down due to low battery.

In fact, such `WakeLock` types have been deprecated, with the last of them being flagged as deprecated in API Level 17. The recommended alternative is to use `android:keepScreenOn` or `setKeepScreenOn()` on some `View`. This will keep the screen on, so long as the activity hosting that `View` is in the foreground. That way, just moving to the background releases the underlying `WakeLock`, allowing the device to return to sleep.

However, in some cases, even that may be insufficient. Suppose that the user is in your activity, and they get distracted, putting down their device for an extended period. Unless you somehow detect the inactivity, and manually turn off the keep-screen-on mode, the screen will stay on indefinitely, until the power is drained. Hence, if you have a decent way of determining if the user is still using your activity, consider using that as a way to determine when the device is inactive (e.g., a `postDelayed()` that gets canceled and rescheduled when the user does something, so if the `postDelayed()` `Runnable` gets invoked, you know the user has done nothing for the delay period). Then, if you know the device is inactive, call `setKeepScreenOn(false)` to return the screen to its normal operating mode.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The academic paper "How is Energy Consumed in Smartphone Display Applications?" has a more extended analysis of screen power draw.

Disk I/O

Disk I/O gets more efficient with bigger operations.

You can see this in something like SQLite, where wrapping a bunch of `INSERT` statements into a single transaction can have substantial benefits in terms of how long the I/O takes.

Not surprisingly, this has a similar impact on power consumption:

- Writing 1GB of data 1,000 bytes at a time is about twice as expensive as is writing it 10,000,000 bytes at a time
- Writing 1GB of data 100 bytes at a time is about five times as expensive as is writing it 1,000 bytes at a time

Hence, you want to try to batch up your disk I/O, where possible, to do fewer, bigger operations, rather than lots of little ones. This includes:

- Batching database I/O in a transaction, as noted above
- Caching data that you intend to log to disk in memory and only writing when your in-memory buffer reaches a certain size or age (though beware the dangers of your process being terminated before you get a chance to write the data)
- Consider using larger buffer sizes with `BufferedInputStream` and `BufferedOutputStream`, if you can afford the heap space, though the 8KB defaults are not that bad

As a rough model, consider disk I/O to draw ~200mA. The smaller the I/O operations, the more time it takes you to accomplish the work, and hence the less efficient those operations are.

While disk I/O is relatively expensive while it is occurring, most apps are not continuously reading or writing, and therefore the total impact to the battery will not be that bad. Apps that *do* continuously use the disk — such as music or video players — will

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

WiFi and Mobile Data

Internet access via WiFi and mobile data networks are another area that you, the developer, tend to control. Some apps require continuous Internet access and only while in the foreground, like a streaming media player. But many more apps wind up doing Internet access periodically in the background, looking for new information on some server somewhere. Unfortunately, these are the sorts of "vampire" apps that can drain the battery without users necessarily being aware of it. Individually, these apps might not even appear all that bad, but when a device has dozens of them, the combined impact results in poor battery life.

Moreover, we also have the problem of dealing with multiple ways of getting to the Internet. Simple solutions will leave us totally oblivious to the differences in downloading via WiFi versus mobile data, at the potential cost in battery consumption. Slightly less-simple solutions optimize for mobile data, to try to minimize power drain in that model. More-elaborate solutions detect what sort of connection we have (using `ConnectivityManager`) and choose among different strategies as connectivity changes.

Here are some things you can do to try to help manage your Internet power consumption.

Use Less

The simplest, rough-cut way to consume less power for Internet access is to do less Internet access in the first place. The less time you spend downloading (or uploading) data, the less power you tend to draw while doing so. In a very coarse approximation, battery consumption will be proportional to bandwidth consumption.

And, of course, consuming less bandwidth can have other benefits, particularly for people on metered mobile data plans.

There are chapters elsewhere in the book that cover ways to deal with bandwidth consumption for bandwidth's sake.

Use What You Already Downloaded

For data that is likely to be unchanging, use a disk cache, so you can avoid downloading the same content again. Such a cache can be used at two levels:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. Simply by having the file in the cache can be a signal to your app that you already have the data and can avoid any sort of request to fetch it again.
2. For HTTP, by recording some additional details (`If-Modified-Since` and `ETag` headers), you can make a request to the server to download the content again, where the server can tell you if you already have the current copy of the content (via a `304` response code).

Many of the Internet libraries discussed earlier in this book offer disk caching as part of their services.

Use In Batches

As noted earlier in this section, in a very coarse approximation, battery consumption will be proportional to bandwidth consumption.

Unfortunately, that approximation is pretty coarse.

We as developers tend to think of Internet access as being like a faucet with two states: on and off. In reality, wireless radios tend to have three states: full power, low power, and standby mode. Opening a socket will bring the radio to full power. An idle radio (no packets transferred) will drop to low power after a while, and eventually back to standby mode. Not surprisingly, the power draw for full power is substantially more than low power, which in turn is more than standby.

However, this model introduces some problems:

- There is some latency to move from standby or low power to full power. This slows down data transfer while the radio "warms up".
- The idle time needed to transition to a lower power state is substantial, with values in the 5–15 second range well within reason. This means that making a request has lingering power cost even *after* our request has completed.

The net is that you want to bring the radio to full power as few times as possible (to minimize the percentage of time we are slowly dropping back to standby and consuming power while we do). And, while we are at full power, we want to do all necessary — or perhaps possibly necessary — data transfers, to avoid having to go back to full power again any time soon.

In other words, you want to batch your network I/O. This is reminiscent of the recommendations to batch disk I/O from earlier in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

So, for example, if you are going to upload data to a server, use that same pulse of work to download anything that needs downloading, rather than having separate schedules for uploads and downloads. Doing more in a batch and having fewer batches will reduce the cost of the power state changes.

Use When the Server Wants You To

One common pattern for Internet access is to poll a server. This is fairly easy to code, using something like `AlarmManager` to get control every so often.

However, this approach resembles children in the back seat of a car, frequently pestering their parents with "Are we there yet?".

Just as the parents will tell the children "We will get there when we get there, and we will tell you when we get there", you can take a similar approach, using Google Cloud Messaging (GCM). Rather than poll the server periodically, have the server contact your app on the device when there is data ready to be downloaded. This works well in cases where polls are likely to result in "yes, we have no data" responses — the pushes can be far less frequent than the polls would be. This can also reduce load on your servers, for not having to respond to poll requests across all your users.

Note, though, that the battery benefits are from using GCM itself. From the standpoint of an app, GCM is "always on", and the power consumed by GCM is attributed to Android itself, not to the app. Hence, pushes are almost "free" from the standpoint of power cost. This will *not* be the case if you "roll your own" push system (MQTT, WebSockets, etc.). In this case, you are attempting to keep a long-lived socket yourself, *in addition* to the one maintained by GCM. Clearly, there are ways to do this that minimize the power consumption of the long-lived socket connection, but that is not easy to accomplish. Hence, you need to weigh the costs of depending upon the Play Services SDK and routing your communications through Google's servers with the costs of trying to do your own separate push mechanism in a battery-friendly fashion.

Use When Android Wants You To

If server push through GCM is impractical (e.g., you do not control the server), you can reduce your power use for Internet access by batching *across apps*, in addition to batching within your app.

What Google wants you to use for synchronizing data with a server is the `SyncManager`. This is an overly-complicated framework that, among other things,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

gives you control to sync to the server at the same time that other apps needing to sync get control. That way, we can "warm up" the wireless radio once and handle several apps' worth of data transfers at once. `SyncManager` will be covered in this book eventually.

Part of the reason why Android moved to make alarms with `AlarmManager` more "inexact" in API Level 19+ is for this same sort of batching. While `AlarmManager` certainly can be used for a variety of purposes, a lot of apps use it for Internet data transfer. Allowing Android to control when those alarms occur allows Android to try to coalesce them, and perhaps even time them to happen when `SyncManager`-led transfers occur, with the objective of minimizing the number of times we bring the wireless radio out of standby mode.

Use Additional Reading

The Android developer documentation has a series of "training" pages on minimizing power consumption for data transfers. This expands upon Reto Meier's Google I|O presentations that touch upon this topic.

GPS

In light testing, GPS seems to draw ~35mA. Additional power will be consumed for using those results, though, and so the net effect on the battery will be somewhat higher, depending upon what your app does when it gets a GPS fix.

That figure is corroborated by the academic paper "An Analysis of Power Consumption in a Smartphone", though that paper tested rather old devices (HTC Dream and Nexus One).

Again, different devices will have different components, and some devices' GPS modules may be more or less efficient.

Hence, GPS itself is a power drain, but not a massive one... if what you are doing with the GPS fixes itself is efficient. Keeping the GPS on for several hours will certainly take a chunk out of the battery charge, but if you are doing lots of work (e.g., navigation app) in response to those fixes, several hours may be more than the battery can handle.

If you can get by with the dependency on the Play Services SDK, using `LocationClient` can help here, particularly in cases where the user may not be

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

moving much, as Google's fused location provider uses the accelerometer to help determine how much they need to use GPS versus other possible means of determining location.

Camera

The camera will consume power while it is actively receiving input, whether that is for the preview frames or for taking full-resolution pictures or video. Of course, it will also consume additional power when recording images to disk, whether those be still photos or continuous video.

A rough guide is that a camera preview will draw ~200mA *plus* the power for screen, CPU, etc. That could easily total over 350mA, even if you are not doing much. Normally, though, the camera preview is on for short periods of time, and only under user control.

A corresponding value for recording video, including the disk I/O and camera preview, would be ~600mA (plus the screen). That is the sort of thing you only want to do in short bursts, as a couple of hours of video recording can really take a bite out of battery. However, once again, normally the user is the one controlling when video is recorded.

Additional Sources

The above sources of power drain are comparatively easy to model and provide a heuristic for determining your possible power usage.

However, there are plenty of other things that can drain the battery, for which this chapter does not provide such a heuristic. In many cases, the usage patterns of the system component will vary so widely that a simple heuristic is unrealistic. In some cases, the power drain from components from different manufacturers will be very different. In some cases, the author of this book simply lacks sufficient expertise with the technology to provide much help (e.g., Bluetooth).

The sections that follow will try to provide some help, though.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

CPU/GPU

Perhaps the biggest source of power drain beyond the components listed above will be the processors: the CPU and the GPU. These draw a fair bit of power, which is why processor manufacturers go to great lengths to try to adapt to varying conditions, turning off cores or switching clock speeds, to try to minimize the power drain.

Usually, so long as we are in the foreground, any CPU/GPU usage impact on power will be considered "normal" by the user. Of course, trying to boost performance here can benefit the user, not only in terms of possibly reduced power consumption, but less lag or other forms of sluggishness. Hence, trying to optimize processor utilization is worthwhile.

However, the bigger complaints from the user will come from power drain while your app is in the background. The biggest source of those complaints will come from your use of `WakeLocks`, preventing the device from going into a low-power sleep state.

There are some apps available on the Play Store that reportedly can give you some idea of how long you may be holding a `WakeLock`, however they generally require root, particularly for Android 4.4+.

Sensors

Sensors, more so that many other device components, seem to get sourced from a wide range of manufacturers. They also seem to be tied into the devices differently from device to device. For example, some devices allow sensors to continue collecting data while the device is otherwise in a sleep mode, while many do not.

As such, it is difficult to give much guidance in terms of power drain tied to your use of sensors.

That being said, here are a few notes that may help:

1. Generally speaking, the more you use a sensor, the more likely it is that it will reflect in power drain. However, only some of that power drain will be from the sensor hardware itself. Your application code processing sensor events will bear much of the blame. Reducing the periods of time when you are registered for sensor events, using longer delays between events, and

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

sensor event batching are ways that you can reduce the power drain associated with the sensors and your associated code.

2. Conversely, in some environments, use of a particular sensor may be "free", insofar as the device uses the sensor itself on a continuous basis. For example, the accelerometer and/or gyroscope is used by devices to detect orientation changes. Hence, those sensors must be powered on regularly, and therefore you cannot be "blamed" for the fact that the sensors are drawing power. Your use of the sensor data may contribute to power drain, of course.

Audio Input and Output

Playing audio through the earpiece, speaker, wired headset, or Bluetooth, will consume some amount of power. The amount will vary by how long you are playing the audio and how the audio is played (e.g., Bluetooth may require more power than on-device audio output). However, in both cases, usually the user has control over the audio, particularly if it is to be playing for a lengthy period of time (e.g., music player), and so the power drain associated with audio playback is less likely to be considered to be a problem, as users will get annoyed with *uncontrolled* power drain, more so than power drain that they they can manage themselves.

Recording audio via the on-board microphone or Bluetooth should also consume some incremental power. In cases where the user is in control over when recording is happening, the power drain is unlikely to cause the user much distress.

Where both playback and recording of audio may cause perceived power problem is in places where the user has less control. For example, an alarm clock app should have some sort of timeout to stop playing the ringtone (or whatever) after some period, if the user fails to respond to the alarm. After all, it is possible that the user is not where the device is and is not in position to stop the alarm. In this case, the power drain will be from several components, audio playback being just one, but it is the uncontrolled nature of the power drain that can get you in trouble.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Addressing Application Size Issues

Sometimes, our apps are just too big, where "too big" can be defined as:

- Bigger than the 50MB limit imposed by the Play Store
- Bigger than some other limit imposed by some other distribution channel
- Big enough that we worry about bandwidth costs, particularly for users on metered data plans
- Big enough that we hit some internal Dalvik limitations

This chapter will review various techniques for trying to keep the size of your app down to a reasonable level.

Prerequisites

This chapter assumes that you have read the core chapters of the book.

Java Code, and the 64K Method Limit

In ordinary Java development, there are few limits as to how big your applications can get. You tend to run into physical limitations, such as available system RAM, before you run into any limitations of the programming language or runtime environment.

And, normally, in Android applications, you do not worry about how many classes or methods you have. However, "normally" is not "always", and there is a specific scenario that complex apps need to worry about.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What Is It?

Quoting Andy Fadden, Android platform engineer:

> The issue is not with the Dalvik runtime nor the DEX file format, but with the current set of Dalvik instructions.
>
> You can reference a very large number of methods in a DEX file, but you can only invoke the first 65536, because that's all the room you have in the method invocation instruction.
>
> I'd like to point out that the limitation is on the number of methods referenced, not the number of methods defined. If your DEX file has only a few methods, but together they call 70,000 different externally-defined methods, you're going to exceed the limit.
>
> [An externally-defined method is] a method defined in a separate DEX file. For most apps this would just be framework and core library / uses-library stuff.

Specifically, you will crash at compile time, with an error message akin to:

```
Unable to execute dex: method ID not in [0, 0xffff]: 65536
Conversion to Dalvik format failed: Unable to execute dex: method ID not
in [0, 0xffff]: 65536
```

64K Seems Like a Lot of Typing…

Well, it is, and it isn't.

First, it is not merely your own methods. You can reach the 64K method limit without implementing 64K methods in your application yourself. You can:

- Call lots of methods defined by the framework
- Absorb lots of methods from libraries, particularly larger libraries that offer many more features than your app uses

This still tends to mean that simpler apps are unlikely to run into this limit, while more complex apps might.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Jake Wharton has published a shell script that can provide you with a count of referenced methods.

Mitigation Tactics

If you are relatively close to the 64K method limit, you may be able to tweak your project to get back under the limit without having to significantly rework your project.

Use Granular Libraries

Some libraries, like Google Play Services, come in two forms: a "kitchen sink" and more granular libraries for individual features. If your need for the library can be met by the granular libraries, use them, and you can remove your dependency on the "kitchen sink".

In the case of Google Play Services, try *not* to depend upon the `com.google.android.gms:play-services` artifact. Instead, try to depend upon one of the more granular artifacts, such as `com.google.android.gms:play-services-maps` for Maps V2. For services, like Google Cloud Messaging, that have no specific granular artifact, depend instead upon `com.google.android.gms:play-services-base` — while still large, this is far smaller than is `com.google.android.gms:play-services`.

Use Better Libraries

One common culprit of hitting the 64K method limit comes from libraries, as their methods count along with yours. Hence, choosing different libraries can perhaps reduce your method count.

One specific case of this comes from the code generated by Google's Protocol Buffers. If you are using Protocol Buffers heavily, your generated classes may each be defining hundreds of unused methods. Switching to an alternative implementation can reduce this significantly. Some such implementations include:

- micro-protobuf
- Square's Wire

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Use ProGuard

If your debug builds are failing due to the 64K method limit, try a release build. If that works, the reason is ProGuard and its ability to strip out code that is deemed to be unreachable.

In this case, you can "buy yourself some time" by arranging to build your app in debug mode with ProGuard, but without ProGuard's normal code obfuscation work (e.g., `-dontoptimize -dontobfuscate` switches in the ProGuard configuration). Quoting Eric Lafortune, ProGuard's lead developer:

> If you apply ProGuard with shrinking enabled but optimization and obfuscation disabled (-dontoptimize -dontobfuscate), the code will already be more compact, and you can still use a debugger. The source files, class names, method names, line numbers, etc remain unchanged and any breakpoints in removed unreachable code are irrelevant.

However, this is not possible with standard Eclipse builds and is difficult to arrange with Ant. With Gradle-based builds, it should be possible to set this up using `minifyEnabled true` and a custom ProGuard configuration file:

```
android {
    buildTypes {
        debug {
            minifyEnabled true
            proguardFile 'proguard-no-obfuscate.txt'
        }
    }
}
```

(where `proguard-no-obfuscate.txt` contains the `-dontoptimize -dontobfuscate` switches)

Mitigation Strategies

If the aforementioned tactics are insufficient — or if they help somewhat, but you are still near the limit with a lot of development yet to be done — you may need to pursue some more strategic ways of resolving your application size.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Don't Go Overboard

One source of method explosion comes from too much adherence to server-side Java coding styles.

For example, if you find yourself defining hundreds of interfaces and/or abstract classes, with `Factory` classes (and perhaps `FactoryFactory` classes), you are more likely to hit the 64K method limit due to all those separate definitions. Consider whether the flexibility that you believe that you obtain from this coding style is worth the risk.

Smaller Apps, Loosely Connected

It may be that you are simply creating an app that is entirely too complicated for the Android environment. Android's `Intent` system is designed to enable apps to inter-operate, and so you may need to consider splitting your app into pieces, such as:

- A suite of related apps
- A host app and plugin apps that enable additional functionality
- An app and an affiliated Web app, where certain functionality is handled by the Web app in a standard browser

Splitting Into Separate DEX Files

It is also possible — though rather risky — to split your app into multiple DEX files. The basic technique is outlined by Google in an Android Developers Blog post.

In Google's formulation, the secondary DEX file is packaged as an application asset and is unpacked into internal storage on first run of your app. This keeps all of your code in your one APK file for distribution.

The risk comes in if you decide to *not* ship the secondary DEX file with the APK, but rather obtain it by other means, such as downloading it from your own Web site. If somebody hacks your Web site, or employs a man-in-the-middle attack when users try downloading the DEX file, your DEX could be replaced by one that contains malware or otherwise harms the user. If you elect to distribute the secondary DEX files yourself by this sort of means, *please* consider the security ramifications and take appropriate steps to ensure that the DEX you download is the unmodified DEX file.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Native Code

Native code, implemented as NDK-compiled libraries, represent another source of app bloat. This will occur regardless of whether the NDK code is yours or if you are using a third-party library that supplies those binaries (e.g., SQLCipher for Android).

Native code is not intrinsically large. However, in some cases, native code is a port from some other environment (or environments) and may contain a lot of stuff that your app does not need. Worse, ProGuard will not strip out unused native code, as its algorithms only work with Java-style bytecode. Hence, it is not out of the question for apps to devote several MB just to the Linux `.so` files that make up the NDK-compiled libraries.

Fortunately, there are some workarounds.

Mitigation via Per-CPU APKs

Some distribution channels, like the Play Store, support publishing multiple versions of an APK, with different versions for different CPU architectures. Hence, you could have one APK with x86 binaries and one APK with ARM binaries, as opposed to having one "fat binary" with both.

While setting this up using the classic build tools would be a major pain, Gradle for Android makes this fairly straight-forward, using product flavors for the CPU architectures:

```
productFlavors {
    x86 {
        ndk {
            abiFilter "x86"
        }
    }
    arm {
        ndk {
            abiFilter "armeabi-v7a"
        }
    }
    mips {
        ndk {
            abiFilter "mips"
        }
    }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Using product flavors this way will give you separate commands for compiling each of the CPU architectures (e.g., `gradle assembleArmRelease`).

More details about using the NDK with Gradle can be found elsewhere in the book.

Mitigation via libhoudini

As is noted in the chapter on the NDK, `libhoudini` is proprietary Intel code that allows ARM-compiled NDK binaries to run on x86 CPUs, using the same sort of opcode translation that is used by the Android emulator. Many, though not all, x86-powered Android devices have `libhoudini`. Those that do could run your app even if you only ship ARM NDK binaries and not x86 ones. This gives you the same sort of space savings as you would get by publishing separate ARM vs. x86 APKs (per the previous section), without having to manage multiple APKs yourself. The cost is speed, as the translation layer adds significant overhead, much as you see with the Android emulator running ARM emulator images instead of x86 ones.

Mitigation via Ignoring Non-ARM

Of course, what a lot of developers do is simply only worry about ARM.

While Google does not publish percentages of CPU architectures the way they do Android OS versions, it is safe to say that, as of early 2014, ~1% of Android devices are powered by non-ARM CPUs. That percentage may climb, particularly as Intel pushes more x86 chipsets. But the *vast* majority of Android devices are powered by ARM. So, even if some of those x86 environments lack `libhoudini` (e.g., the manufacturer did not license `libhoudini` from Intel), they are so few in number that developers are prone to ignore x86.

Ironically, what drives x86 for developers is the development environment itself, not the production environment. The x86 emulator is nicely responsive, compared to a similarly-configured ARM emulator image. Many developers eschew the ARM emulator entirely, with it being too slow. Hence, developers may be interested in having x86 binaries in the APK to allow the app to run on the x86 emulator (which lacks `libhoudini`). In this case, it may be worthwhile to have a dedicated release build process that strips out the x86 binaries, if the space that those binaries take up is more than you can afford.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Images

Bitmap images are notorious for taking up lots of heap space. However, they can also swell the size of your APK. While the bitmap PNG or JPEG files will be compressed on disk, if you have enough of them, they can still consume many MB of space in the APK, particularly since the APK cannot compress them further.

Mitigation via Resource Aliases

You may have multiple copies of the same image.

The example cited in the Android documentation is where you want to have locale-specific drawable images. For example, perhaps you want to show a flag, and you use language resource sets to try to map the right flag to the right language. However, some flags are going to be used in multiple languages, such as the Canadian flag being needed for `en-rCA` and `fr-rCA`. By default, you would place your flag icon in each of those resource sets, duplicating your results. This gets worse if you have a few versions of the same flag icon for different densities.

However, you can elect to use a resource aliases to handle this differently.

Suppose that your code refers to a `flag` drawable resource (e.g., `@drawable/flag` or `R.drawable.flag`). For many languages, you would have a unique flag in the appropriate resource set. For cases where the same flag is used in multiple situations:

1. Put the flag in a resource set that is not tied to locale (e.g., `res/drawable-hdpi/` instead of `res/drawable-en-rCA-hdpi/`), for as many densities as you choose, but under a different name (e.g., `flag_canada.png` instead of `flag.png`)
2. Create a small XML file, `flag.xml`, in each of the locale-specific directories (e.g., `res/drawable-en-rCA/` and `res/drawable-fr-rCA/`), pointing to your `flag_canada` drawable:

```
<?xml version="1.0" encoding="utf-8"?>
<bitmap
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:src="@drawable/flag_canada" />
```

When you reference `R.drawable.flag` on an `en-rCA` or `fr-rCA` device, Android will read in the XML resource, then turn around and retrieve the `flag_canada` drawable,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

and use that. Since the two XML files are likely to be smaller than the sum total of the duplicate copies, you save disk space.

Mitigation via pngquant

In practice, the above technique is just not that commonly used, because it addresses a fairly narrow scenario. A more general-purpose solution is to try to tweak the images to be visually nearly identical, yet take up less disk space.

There are a variety of tools for this, mostly aimed at Web development, where smaller image file sizes means faster-loading Web pages.

One such tool is `pngquant`. Given a PNG file as input, it generates a smaller PNG file as output, one with an optimized color palette, using mathematical techniques to choose colors that will maintain as much of the original look as possible. Many of the images in this book were optimized using `pngquant`, at a substantial savings in disk size, without materially sacrificing image quality.

APK Expansion Files

The ultimate solution to disk space concerns, for distribution through the Play Store, is to get stuff out of your app entirely and distribute that stuff by other means. The Play Store offers APK expansion files with this in mind. You can publish one or two expansion files, each containing up to 2GB of files. While these will not be treated as resources or assets, you do have access to the file contents at runtime. Game developers will use these for sound effects, additional artwork, and so on. The biggest limitation is that these files may not be supported by all distribution channels.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Alternatives for App Development

Special Creative Commons BY-NC-SA 4.0 License Edition

The Role of Alternative Environments

You might think that Android is all about Java. The official Android Software Development Kit (SDK) is for Java development, the build tools are for Java development, the discussion groups and blog posts and, yes, most books are for Java development. Heck, most of this book is about Java.

However (and with apologies to William Goldman), it just so happens that Android is only mostly Java. There's a big difference between mostly Java and all Java. Mostly Java is slightly not Java.

So, while Android's "sweet spot" will remain Java-based applications for the near term, you can still create applications using other technologies. This part of the book will take a peek at some of those alternatives.

This chapter starts with an examination of the pros and cons of Android's Java-centric strategy. It then enumerates some reasons why you might want to use something else for your Android applications. The downsides of alternative Android application environments – lack of support and technical challenges – are also discussed.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In the Beginning, There Was Java…

The core Android team made a fairly reasonable choice of language when they chose Java. It is a very popular language, and in the mobile community it had a clear predecessor in Java Micro Edition (J2ME). Lacking direct access to memory addresses (so-called "pointers"), a Java-based application will be less prone to developer errors leading to buffer overruns, resulting in possible hacks. And there is a fairly robust ecosystem around Java, in terms of educational materials, existing code bases, integrated development environments (IDEs), and so on.

However, while you can program Android in the Java language, an Android device does not run a Java application. Instead, your Java code is converted into something that runs on the "Dalvik virtual machine". This is akin to the technology used for regular Java applications, but Dalvik is specifically tuned for Android's environment. Moreover, it limits the dependency of Android on Java itself to a handful of programming tools, important as Java's stewardship moves from Sun to Oracle to wherever.

That Dalvik virtual machine is also capable of running code from other programming languages, a feature that makes possible much of what this book covers.

… And It Was OK

No mobile development environment is perfect, and so the combination of Java and Android has its issues.

Java uses garbage collection to save people from having to keep track of all of their memory allocations. That works for the most part, and it is generally a boon to developer productivity. However, it is not a cure-all for every memory and resource allocation problem. You can still have what amounts to "memory leaks" in Java, even if the precise mechanics of those leaks differ from the classic leaks you get in C, C++, etc.

Most importantly, though, not everybody likes Java. It could be because they lack experience with it, or perhaps they have experience with it and did not enjoy that experience. Certainly, Java is slowly being considered as a language for big enterprise systems and, therefore, is not necessarily "cool". Advocates of different languages will have their own pet peeves with Java as well (e.g., to a Ruby developer, Java is really verbose).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

So, while Java was not a bad choice for Android, it was not perfect, either.

Bucking the Trend

However, just because Java is the dominant way to build apps for Android, that does not mean it is the only way, and for you, it may not even be the best way.

Perhaps Java is not in your existing skill set. You might be a Web developer, more comfortable with HTML, CSS, and JavaScript. There are frameworks to help you with that. Or, maybe you cut your teeth on server-side scripting languages like Perl or Python — there are ways to sling that code on Android as well. Or perhaps you already have a bunch of code in C/C++, such as game physics algorithms, that would be painful to rewrite in Java — you should be able to reuse that code too.

Even if you would be willing to learn Java, it may be that your inexperience with Java and the Android APIs will just slow you down. You might be able to get something built much more quickly with another framework, even if you wind up replacing it with a Java-based implementation in the future. Rapid development and prototyping is frequently important, to get early feedback with minimal investment in time.

And, of course, you might just find Java programming to be irritating. You would not be the first, nor the last, to have that sentiment. Particularly if you are getting into Android as a hobby, rather than as part of your "day job", having fun will be important to you, and you might not find Java to be much fun.

Support, Structure

However, "friendly" and "fully supported" are two different things.

Some alternatives to Java-based development are officially supported by the core Android team, such as C/C++ development via the Native Development Kit (NDK) and Web-style development via HTML5.

Some alternatives to Java-based development are supported by companies. Adobe supports AIR, Nitobi supports PhoneGap, Rhomobile supports Rhodes, and so on. Other alternatives are supported by standards bodies, like the World Wide Web Consortium (W3C) supporting HTML5. Still others are just tiny projects with only the backing of a couple of developers.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You will need to make the decision for yourself which of these levels of support will meet your requirements. For many things, support is not much of an issue, but there will always be cases where support becomes paramount (e.g., enterprise application development).

Caveat Developer

Of course, going outside the traditional Java environment for Android development has its issues, beyond just how much support might be available.

Some may be less efficient, in terms of processor time, memory, or battery life, than will development in Java. C/C++, on the whole, is probably better than Java, but HTML5 may be worse, for example. Depending on what you are writing and how heavily it will be used will determine how critical that inefficiency will be.

Some may not be available on all devices. Right now, Flash is the best example of this — some devices offer some amount of Flash support, while other devices have no Flash at all. Similarly, HTML5 support was only added to Android in Android 2.0, so devices running older versions of Android do not have HTML5 as a built-in option.

Every layer between you and officially supported environments makes it that much more difficult for you to ensure compatibility with new versions of Android, when they arise. For example, if you create an application using PhoneGap, and a new Android version becomes available, there may be incompatibilities that only the PhoneGap team can address. While they will probably address those quickly — and they may provide some measure of insulation to you from those incompatibilities — the response time is outside of your control. In some cases, that is not a problem, but in other cases, that might be bad for your project.

Hence, just because you are developing outside of Java does not mean everything is perfect. You simply have to trade off between these problems and the ones Java-based development might cause you. Where the balance lies is up to each individual developer or firm.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

HTML5

Prior to the current wave of interest in mobile applications, the technology *du jour* was Web applications. A lot of attention was paid to AJAX, Ruby on Rails, and other techniques and technologies that made Web applications climb close to the experience of a desktop application, and sometimes superior.

The explosion of Web applications eventually drove the next round of enhancements to Web standards, collectively called HTML5. Android 2.0 added the first round of support for these HTML5 enhancements. Notably, Android supports offline applications and Web storage, meaning that HTML5 becomes a relevant technique for creating Android applications, without dealing with Java.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate. Reading the chapter on `WebView` would be a good idea, as would reading the introduction to this trail.

Offline Applications

The linchpin for using HTML5 for offline applications — on Android or elsewhere — is the ability for those applications to be used when there is no connectivity, either due to problems on the client side (e.g., on an airplane sans WiFi) or on the server side (e.g., Web server maintenance).

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What Does It Mean?

Historically, Web applications have had this annoying tendency to require Web servers. This led to all sorts of workarounds for offline use, up to and including shipping a Web server and deploying it to the desktop.

HTML5 solves this problem by allowing Web pages to specify their own caching rules. A Web app can publish a "cache manifest", describing which resources:

1. Can be safely cached, such that if the Web server is unavailable, the browser will just use the cached copy
2. Cannot be safely cached, such that if the Web server is unavailable, the browser should fail like it normally does
3. Have a "fallback" resource, such that if the Web server is unavailable, the cached fallback resource should be used instead

For mobile devices, this means that a fully HTML5-capable browser should be able to load all its assets up front and keep them cached. If the user loses connectivity, the application will still run. In this respect, the Web app behaves almost identically to a regular app.

How Do You Use It?

For this chapter, we will use the Checklist "mini app" created by Alex Gibson. While the most up-to-date version of this app can be found on Mr. Gibson's Web site, this chapter will review the copy found in `HTML5/Checklist`. This copy is also hosted online on the CommonsWare site, or via a shortened URL: `http://bit.ly/cw-html5`.

About the Sample App

Checklist is, as the name suggests, a simple checklist application. When you first launch it, the list will be empty:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

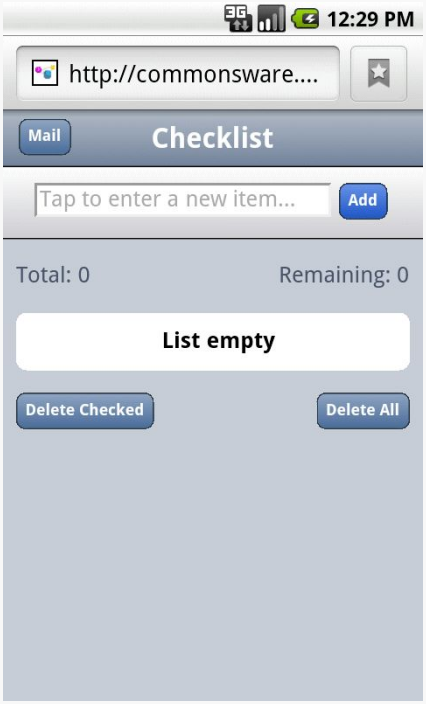


Figure 814: The Checklist, as initially launched

You can enter some text in the top field and click the Add button to add it to the list:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

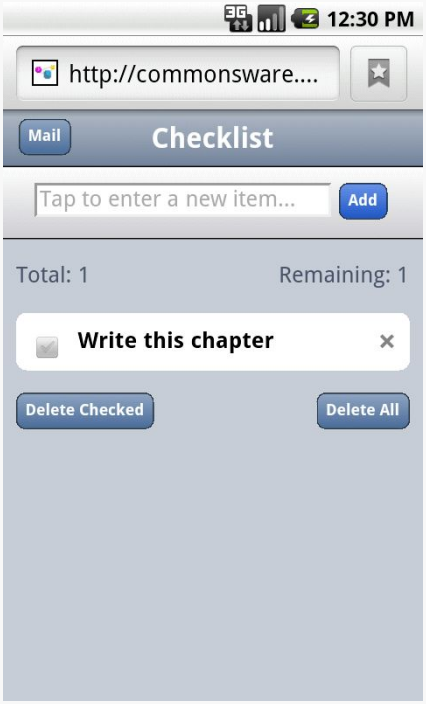


Figure 815: The Checklist, with one item added

You can "check off" individual items, which are then displayed in strike-through:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

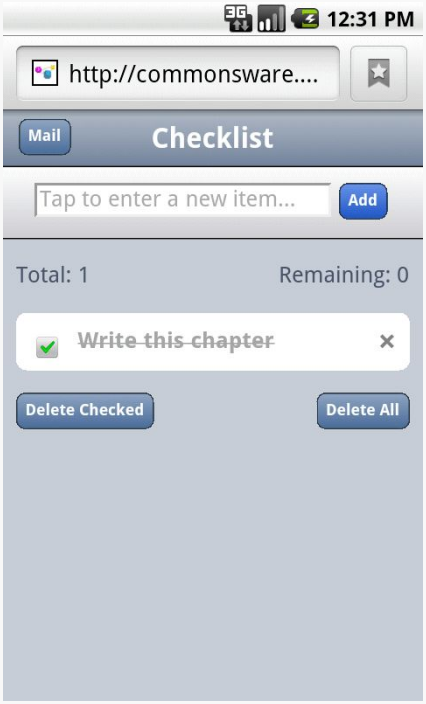


Figure 816: The Checklist, with one item marked as completed

You can also delete the checked entries (via the Delete Checked button) or all entries (via the Delete All button), which will pop up a confirmation dialog before proceeding:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

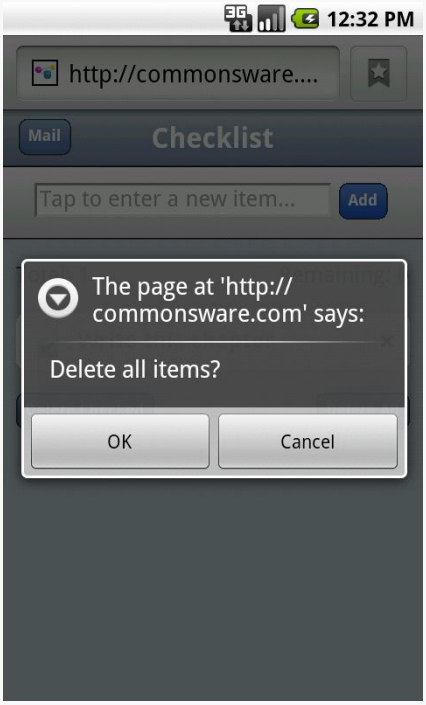


Figure 817: The Checklist's delete confirmation dialog

"Installing" Checklist on Your Phone

To access Checklist on your Android device, visit one of the URLs above for the hosted edition using the Browser application — the shortened one may be easiest to enter into the browser on the device. You can then add a bookmark for it (More > Add bookmark from the browser's options menu) to come back to it later.

You can even set up a shortcut for the bookmark on your home screen, if you so choose — just long-tap on the background, choose Bookmark, then choose the Checklist bookmark you set up before.

Examining the HTML

All of that is accomplished using just a handful of lines of HTML:

```
<!DOCTYPE html>
<html lang="en" manifest="checklist.manifest">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Checklist</title>
<meta name="viewport"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  content="width=device-width; initial-scale=1.0; maximum-scale=1.0;
user-scalable=0;" />
<meta name="apple-mobile-web-app-capable" content="yes" />
<meta name="apple-mobile-web-app-status-bar-style" />
<link rel="apple-touch-startup-image" href="splashscreen.png" />
<link rel="stylesheet" href="styles.css" />
<link rel="apple-touch-icon-precomposed"
      href="apple-touch-icon-precomposed.png" />
</head>
<body>
<section>
    <header>
      <button type="button" id="sendmail">Mail</button>
      <h1>Checklist</h1>
    </header>
    <article>
       <form id="inputarea" onsubmit="addNewItem()">
          <input type="text" name="name" id="name" maxlength="75"
                 autocorrect placeholder="Tap to enter a new item&hellip;" />
          <button type="button" id="add">Add</button>
        </form>
        <ul id="maillist">
        <li class="empty"><a href="" id="maillink">Mail remaining items</a></li>
      </ul>
        <p id="totals"><span id="tally1">Total: <span id="total">0</span></span>
        <span id="tally2">Remaining: <span id="remaining">0</span></span></p>
      <ul id="checklist">
        <li class="empty">Loading&hellip;</li>
      </ul>
    </article>
    <fieldset>
      <button type="button" id="deletechecked">Delete Checked</button>
      <button type="button" id="deleteall">Delete All</button>
    </fieldset>
</section>
<script src="main.js"></script>
</body>
</html>
```

For the purposes of offline applications, though, the key is the `manifest` attribute of our `html` element. Here, we specify the relative path to a manifest file, indicating what the rules are for caching various portions of this application offline.

Examining the Manifest

So, since the manifest is where all the fun is, here is what Checklist's manifest looks like:

```
CACHE MANIFEST
#version 54
styles.css
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
main.js
splashscreen.png
```

The HTML5 manifest format is extremely simple. It starts with a `CACHE MANIFEST` line, followed by a list of files (technically, relative URLs) that should be cached. It also supports comments, which are lines beginning with `#`.

The manifest can also have a `NETWORK:` line, followed by relative URLs that should never be cached. Similarly, the manifest can have a `FALLBACK:` line, followed by pairs of relative URLs: the URL to try to fetch off the network, followed by the URL of a cached resource to use if the network is not available.

In principle, the manifest should request caching for everything that the application needs to run, though the page that requested the caching (`index.html` in this case) is also cached.

Web Storage

Caching the HTML5 application's assets for offline use is all well and good, but that will be rather limiting on its own. In an offline situation, the application would not be able to use AJAX techniques to interact with a Web service. So, if the application is going to be able to store information, it will need to do so on the browser itself.

Google Gears and related tools pioneered this concept and blazed the trail for what is now variously called "Web Storage" or "DOM Storage" for HTML5 applications. An HTML5 app can store data persistently on the client, within client-imposed limits. That, in conjunction with offline asset caching, means an HTML5 application can deliver far more value when it lacks an Internet connection, or for data that just does not make sense to store "in the cloud".

Note that, technically, Web Storage is not part of HTML5, but is a related specification. However, it tends to get "lumped in with" HTML5 in common conversation.

What Does It Mean?

On a Web Storage-enabled browser, your JavaScript code will have access to a `localStorage` object, representing your application's data. More accurately, each "origin" (i.e., domain) will have a distinct `localStorage` object on the browser.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `localStorage` object is an "associative array", meaning you can work with it either via numerical indexes or string-based keys at your discretion. Values typically are strings. You can:

1. Find out how many entries are in the array via `length()`
2. Get and set items by key via `getItem()` and `setItem()`
3. Get the key for a numerical index via `key()`
4. Remove individual entries via `removeItem()` or remove all items via `clear()`

This means you do not have the full richness of a SQL database, like you might have with SQLite in a native Android application. But, for many applications, this should suffice.

How Do You Use It?

Checklist stores the list items as keys in the associative array, with a value of `0` for a regular item and `1` for a deleted item. Here, we see the code for putting a new item into the checklist:

```
try {
  localStorage.setItem(strippedString, data);
}
catch (e) {
  if (e == QUOTA_EXCEEDED_ERR) {
    alert('Quota exceeded!');
  }
}
```

Here is the code where those items are pulled back out of storage and put into an array for sorting and, later, display as DOM elements on the Web page itself:

```
/*get all items from localStorage and push them one by one into an
array.*/
for (i = 0; i <= listlength; i++) {

  var item = localStorage.key(i);
  myArray.push(item);
}

/*sort the array into alphabetical order.*/
myArray.sort();
```

When the user checks the checkmark next to an item, the storage is updated to toggle the checked setting persistently:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
/*toggle the check flag.*/
if (target.previousSibling.checked) {
  data = 0;
}
else {
  data = 1;
}
/*save item in localStorage.*/
try {
  localStorage.setItem(name, data);
} catch (e) {

  if (e == QUOTA_EXCEEDED_ERR) {
    alert('Quota exceeded!');
  }
}
```

Checklist also has code to delete items from storage, either all those marked as checked:

```
/*remove every item from localStorage that has the data flag checked.*/
while (i <= localStorage.length-1) {

  var key = localStorage.key(i);
  if (localStorage.getItem(key) === '1') {
    localStorage.removeItem(key);
  }
  else { i++; }
}
```

... or all items:

```
/*deletes all items in the list.*/
deleteAll: function() {

  /*ask for user confirmation.*/
  var answer = confirm("Delete all items?");

  /*if yes.*/
  if (answer) {

    /*remove all items from localStorage.*/
    localStorage.clear();
    /*update view.*/
       checklistApp.getAllItems();
   }
   /*clear up.*/
   delete checklistApp.deleteAll;
},
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Web SQL Database

Android's built-in browser also supports a "Web SQL Database" option, one where you can use SQLite-style databases from JavaScript. This adds a lot more power than basic Web Storage, albeit at a complexity cost. It is also not part of an active standard — the WHATWG team working on this standard has set it aside for the time being.

You might consider evaluating Lawnchair, which is a JavaScript API that allows you to store arbitrary JSON-encoded objects. It will use whatever storage options are available, and therefore will help you deal with cross-platform variety. In particular, it supports the Google Gears facility found in some older versions of Android.

Going To Production

Creating a little test application requires nothing magical. Presumably, though, you are interested in others using your application – perhaps many others. Classic Java-based Android applications have to deal with testing, having the application digitally signed for production, distribution through various channels (such as the Android Market), and updates to the application by one means or another. Those issues do not all magically vanish because HTML5 is used as the application environment. However, HTML5 does change things significantly from what Java developers have to do.

Testing

Since HTML5 works in other browsers, testing your business logic could easily take advantage of any number of HTML and JavaScript testing tools, from Selenium to QUnit to Jasmine.

For testing on Android proper — to ensure there are no issues related to Android's browser implementation — you can use Selenium's Android Driver or Remote Control modes.

Signing and Distribution

Unlike native Android applications, you do not need to worry about signing your HTML5 applications.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The downside of this is that there is no support for distribution of HTML5 applications through the Play Store, which today only supports native Android apps. Users will have to find your application by one means or another, visit it in the browser, bookmark the page, and possibly create a home screen shortcut to that bookmark.

Updates

Unlike native Android applications, which by default must be updated manually, HTML5 applications will be transparently updated the next time they run the app while connected to the Internet. The offline caching protocol will check the Web server for new editions of files before falling back to the cached copies. Hence, there is nothing more for you to do other than publish the latest Web app assets.

Issues You May Encounter

Unfortunately, nothing is perfect. While HTML5 may make many things easier, it is not a panacea for all Android development problems.

This section covers some potential areas of concern you will want to consider as you move forward with HTML5 applications for Android.

Android Device Versions

Not all Android devices support HTML5 — only those running Android 2.x or higher. Ideally, therefore, you do a bit of "user-agent sniffing" on your Web server and redirect older Android users to some other page explaining the limitations in their device.

Here is the user-agent string for a Nexus One device running Android 2.1:

```
Mozilla/5.0 (Linux; U; Android 2.1-update1; en-us; Nexus One
Build/ERE27) AppleWebKit/530.17 (KHTML, like Gecko) Version/4.0 Mobile
Safari/530.17
```

As you can see, it is formatted like a typical modern user-agent string, meaning it is quite a mess. It does indicate it is running `Android 2.1-update1`.

Eventually, somebody will create a database of user-agent strings for different device models, and from there we can derive appropriate regular expressions or similar algorithms to determine whether a given device can support HTML5 applications.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Screen Sizes and Densities

HTML5 applications can be run on a wide range of screen sizes, from QVGA Android devices to 1080p LCDs and beyond. Similarly, screen densities may vary quite a bit, so while a 48x48 pixel image on a smartphone may be an appropriate size, it may be too big for a 1080p television, let alone a 24" LCD desktop monitor.

Other than increasing the possible options on the low end of screen sizes, none of this is unique to Android. You will need to determine how best to design your HTML and CSS to work on a range of sizes and densities, even if Android were not part of the picture.

Limited Platform Integration

HTML5, while offering more platform integration than ever before, does not come close to covering everything an Android application might want to be able to do. For example, an ordinary HTML5 application cannot:

1. Launch another application
2. Work with the contacts database
3. Raise a notification
4. Do work truly in the background (though "Web workers" may alleviate this somewhat someday)
5. Interact with Bluetooth devices
6. Record audio or video
7. Use the standard Android preference system
8. Use speech recognition or text-to-speech
9. And so on

Many applications will not need these capabilities, of course. And, one can expect that other application environments, like PhoneGap, will evolve into "HTML5 Plus" for Android. That way, you could create a stock application that works across all devices and an enhanced Android application that leverages greater platform integration, at the cost of some additional amount of programming.

Performance and Battery

There has been a nagging concern for some time that HTML-based user interfaces are inefficient compared to native Android UIs, in terms of processor time, memory,

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

and battery. For example, one of the stated reasons for avoiding BONDI-style Web widgets for the Android home screen is performance.

Certainly, it is possible to design HTML5 applications that will suck down the battery. For example, if you have a hunk of JavaScript code running every second indefinitely, that is going to consume a fair amount of processor time. However, outside of that, it seems unlikely that an ordinary application would be used so heavily as to materially impact battery life. Certainly, more testing will need to be done in this area.

Also, an HTML5 application may be a bit slower to start up than are other applications, if the Browser has not been used in a while, or if the network connection is there but has minimal bandwidth to your server.

Look and Feel

HTML5 applications can certainly look very slick and professional – after all, they are built with Web technologies, and Web apps can look very slick and professional.

However, HTML5 applications will not necessarily look like standard Android applications, at least not initially. Some enterprising developers will, no doubt, create some reusable CSS, JavaScript, and images that will, for example, mirror an Android native `Spinner` widget (a type of drop-down control). Similarly, HTML5 applications will tend to lack options menus, notifications, or other UI features that a native Android application may well use.

This is not necessarily bad. Considering the difficulty in creating a very slick-looking Android application, HTML5 applications may tend to look better than their Android counterparts. After all, there are many more people skilled in creating slick Web apps than are skilled in creating slick Android apps.

However, some users may complain about the look-and-feel disparity, just because it is different.

Distribution

HTML5 applications can be trivially added to a user's device — browse, bookmark, and add a shortcut to the home screen.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, HTML5 applications will not show up in the Play Store, so users trained to look at the Market for available applications will not find HTML5 applications, even ones that may be better than their native counterparts.

It is conceivable that, someday, the Play Store will support HTML5 applications. It is also conceivable that, someday, Android users will tend to find their apps by means other than searching the Android Market, and will be able to get their HTML5 apps that way. However, until one of those becomes true, HTML5 applications may be less "discoverable" than their native equivalents.

HTML5: The Baseline

HTML5 is likely to become rather popular for conventional application development. It gives Web developers a route to the desktop. It may be the only option for Google's Chrome OS. And, with ever-improving support on popular mobile devices — Android among them — developers will certainly be enticed by another round of "write once, run anywhere" promises.

It is fairly likely that, over time, HTML5 will be the #2 option for Android application development, after the conventional Java application written to the Android SDK. That will make HTML5 the baseline for comparing alternative Android development options — not only will those options be compared to using the SDK, they will be compared to using HTML5.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

PhoneGap

PhoneGap is perhaps the original alternative application framework for Android, arriving on the scene in early 2009. PhoneGap is open source, backed by Adobe, who in 2011 acquired Nitobi, the firm founded by PhoneGap's creators.

Prerequisites

Understanding this chapter requires that you have read the chapter on `WebView` and the chapter on HTML5.

What Is PhoneGap?

As the PhoneGap About page puts it:

> Mobile development is a mess. Building applications for each device — iOS, Android, Windows Phone and more — requires different frameworks and languages. One day, the big players in mobile may decide to work together and unify third-party app development processes. Until then, PhoneGap will use standards-based web technologies to bridge web applications and mobile devices. Plus, because PhoneGap apps are standards compliant, they're future-proofed to work with browsers as they evolve.

PhoneGap, today, focuses on bridging the gap between Web technologies and native mobile development, with access to more features than HTML5 applications have.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What Do You Write In?

A PhoneGap application is made up of HTML, CSS, and JavaScript, no different than a mobile Web site or HTML5 application, except that the Web assets are packaged with the application, rather than downloaded on the fly.

A pre-installed PhoneGap application, therefore, can contain comparatively large assets, such as complex JavaScript libraries, that might be too slow to download over slower EDGE connections. However, PhoneGap will still be limited by the speed of mobile devices and how quickly WebKit can load and process those assets.

Also, development for WebKit-for-mobile has its differences over development for WebKit-for-desktops, particularly with respect to touch versus mouse events. You may want to develop using mobile layers of JavaScript frameworks (e.g., jQT versus plain jQuery) where practical.

What Features Do You Get?

As with an HTML5 application, you get the basic capabilities of a Web browser, including AJAX support. Beyond that, PhoneGap adds a number of JavaScript APIs to allow you to get at the underlying features of the Android platform. At the time of this writing, that includes:

1. Accelerometer access, for detecting movement of the device
2. Audio recording
3. Camera access, for taking still pictures
4. Database access, both to databases of your creation (SQLite) or others built into Android (e.g., contacts)
5. File system access, such as to the SD card or other external storage
6. Geolocation, for determining where the device is
7. Vibration, for shaking the phone (e.g., force-feedback)

Since some of these are part of the HTML5 specification (e.g., geolocation), you have your choice of APIs. Also, this list changes over time, so you may have access to more than what is described here.

What Do Apps Look Like?

They will look like Web pages, more so than native Android apps. You can use CSS and images to mimic the Android look and feel to some extent, but only for those

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

sorts of widgets that are readily able to be created in both Android and HTML. For example, the Android `Spinner` widget — which resembles a drop-down list — may be difficult to mimic in HTML.

Here is a screenshot of a PhoneGap example application:

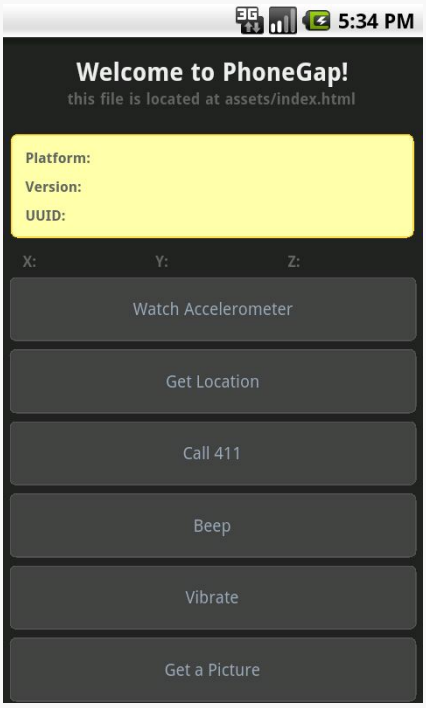


Figure 818: A PhoneGap example application

How Does Distribution Work?

Distributing a PhoneGap application is pretty much identical to distributing any other standard Android application. After testing, you will create a standard APK file with the Android build tools, from an Android project generated for you by PhoneGap. This project will contain the Java, XML, and other necessary bits to wrap around your HTML, CSS, and JavaScript to make up your application. Then, you digitally sign the application and upload it to the Play Store or any other distribution mechanism you wish to use.

What About Other Platforms?

PhoneGap is not just for Android. You can create PhoneGap applications for iOS, Blackberry, some flavors of Symbian, and more. In theory, at least, you can create

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

one application using HTML, CSS, JavaScript, and the PhoneGap JavaScript APIs, and have it run across many devices.

There are a couple of limitations that will hamper your progress to that goal:

- The Web browsing component used by PhoneGap across all those platforms is not identical. Even multiple platforms using WebKit will have different WebKit releases, based upon what was available when WebKit was integrated into a given device's firmware. Hence, you will want to test and ensure your CSS, in particular, works as you would expect on as many devices as possible.
- Not all PhoneGap JavaScript APIs are available on all devices as yet, due to a variety of factors (e.g., not exposed in the platform's native APIs, lack of engineering time to hoist the capability into the PhoneGap APIs). There is a table on the PhoneGap site that will keep you apprised of what works and what does not across the devices. You will want to restrict your feature use to match your desired platforms, or restrict your platforms to match your desired features.

How Is It Licensed?

PhoneGap is available under the Apache Software License 2.0. In 2011, Nitobi contributed PhoneGap to the Apache Software Foundation (ASF) for independent management, just prior to being acquired by Adobe. This has now turned into Apache Cordova.

Using PhoneGap

Now, let's look at more of the mechanics for using PhoneGap.

PhoneGap's installation and usage, as of the time of this writing, normally requires an expert in Java-based Android development. You need to install a whole bunch of tools, edit configuration files by hand, and so forth. If you want to do all of that, documentation is available on the PhoneGap site.

If you are reading this chapter, there's a decent chance that you would rather skip all of that. Hence, for many, the best answer is the PhoneGap/Build service.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Installation

The PhoneGap Web site will allow you to download the latest PhoneGap tools as a ZIP archive. You can unpack those wherever it makes sense for your development machine and platform.

For Android development, that is all of the PhoneGap-specific installation you will need. However, you will need the Android SDK and related tools (e.g., Eclipse, if you wish to use Eclipse) for setting up the project.

Creating and Installing Your Project

A PhoneGap Android project is, at its core, a regular Android project, which you can create following the instructions outlined earlier in this book. To convert the standard generated "Hello, World" application into a PhoneGap project, you need to do the following:

- From the `Android/` directory of wherever you unZIPped the PhoneGap ZIP file, copy the PhoneGap JAR file to the `libs/` directory of your project. If you are using Eclipse, you will also need to add it to your build path.
- Create an `assets/www/` directory in your project. Then, copy over the PhoneGap JS file from the `Android/` directory of wherever you unZIPped the PhoneGap ZIP file.
- Adjust the standard "Hello, World" activity to inherit from `DroidGap` instead of `Activity`. This will require you to import `com.phonegap.DroidGap`.
- In your activity's `onCreate()` method, replace `setContentView()` with `super.loadUrl("file:///android_asset/www/index.html");`
- In your manifest, add all of the permissions that PhoneGap requests, listed later in this chapter.
- Also in your manifest, add a suitable `<supports-screens>` element based upon what screen sizes you are willing to test and support.
- Also in your manifest, add `android:configChanges="orientation|keyboardHidden"` to your `<activity>` element, as `DroidGap` handles orientation-related configuration changes

At this point, you can create an `assets/www/index.html` file in your project and start creating your PhoneGap application using HTML, CSS, and JavaScript. You will need to have a reference to the PhoneGap JavaScript file (e.g., `<script type="text/javascript" charset="utf-8"`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

`src="phonegap.0.9.4.js" />`). When you want to test the application, you can build and install it like any other Android application (e.g., `ant clean debug install` if you are using the command line build process).

For somebody experienced in Android SDK development, setting this up is not a big challenge.

PhoneGap/Build

PhoneGap/Build is a Tools-as-a-Service (TaaS) hosted approach to creating PhoneGap projects. This way, all of the Android build process is handled for you by PhoneGap-supplied servers. You just focus on creating your HTML, CSS, and JavaScript as you see fit.

When you log into PhoneGap/Build, you are first prompted to create your initial project, by supplying a name and the Web assets to go into the app:

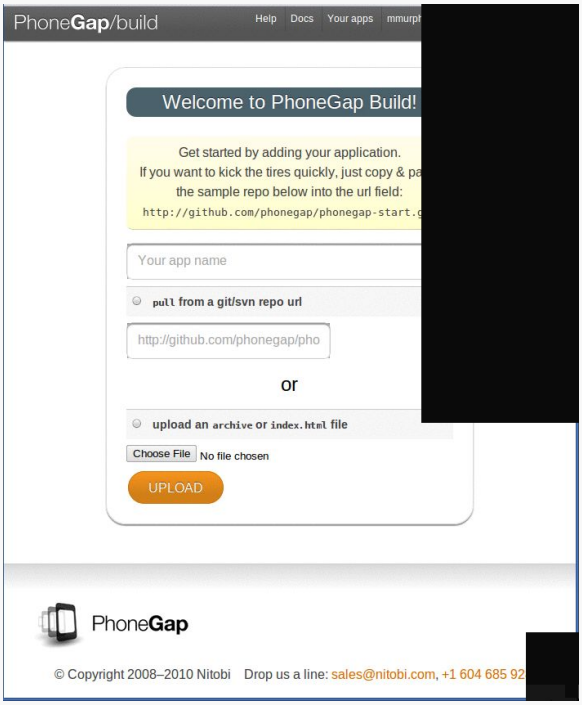


Figure 819: Creating your first project in PhoneGap/Build

You will be able to add new projects later on via a New App button, which gives you the same set of options.

Subscribe to updates at https://commonsware.com
Special Creative Commons BY-NC-SA 4.0 License Edition

Your choices for the assets are to upload a ZIP file containing all of them, or to specify the URL to a public GitHub repository that PhoneGap/Build can pull from. The latter tends to be more convenient, if you are used to using Git for version control, and if your project is open source (and therefore has a public repository).

Once you click the Upload button, the PhoneGap/Build server will immediately start building your application for Android, plus Blackberry, Symbian, and WebOS:

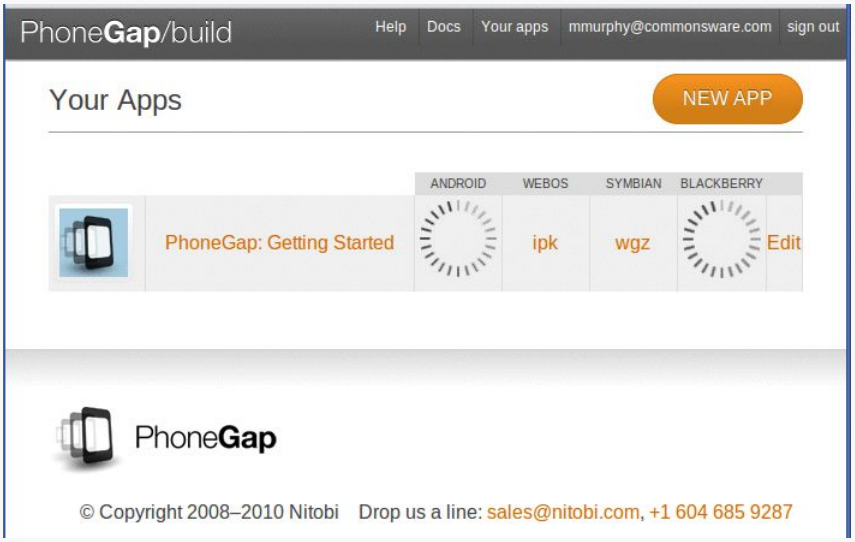


Figure 820: Building your first project in PhoneGap/Build

Each of the targets has its own file extension (e.g., apk for Android). Clicking that link will let you download that file. Or, click on the name of the project, and you get QR codes to enable downloads straight to your test device:

Subscribe to updates at https://commonsware.com
Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 821: Your project's QR codes in PhoneGap/Build

This page also gives you a link to update the app from its GitHub repo (if you chose that option). Or, click Edit to specify more options, such as the version of your application or its launcher icon:

Subscribe to updates at https://commonsware.com
Special Creative Commons BY-NC-SA 4.0 License Edition

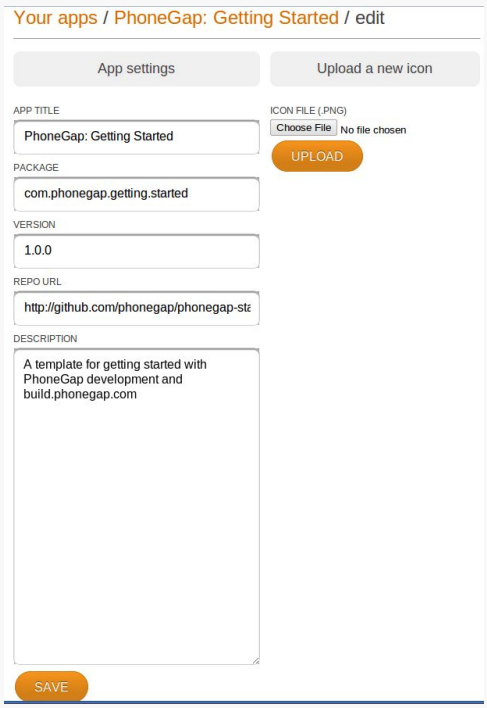


Figure 822: Your project's settings in PhoneGap/Build

All in all, if you do not otherwise need the Android SDK and related tools on your development machine, PhoneGap/Build certainly simplifies the PhoneGap building process.

PhoneGap/Build is free for open source (public) projects, but there are fees associated with private use beyond a single app.

PhoneGap and the Checklist Sample

The beauty of PhoneGap is that it wraps around HTML, CSS, and JavaScript. In other words, you do not have to do much of anything PhoneGap-specific to be able to take advantage of PhoneGap delivering to you an APK suitable for installation on an Android device. That being said, PhoneGap does expose more stuff to you than you can get from the standards, if you need them and are willing to use proprietary PhoneGap APIs for them.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Sticking to the Standards

Given an existing HTML5 application, all you need to do to make it be an installable APK is wrap it in PhoneGap.

For example, to convert the HTML5 version of Checklist into an APK file, you need to:

- Follow the steps to create an empty PhoneGap project outlined earlier in this chapter
- Copy the HTML, CSS, JavaScript, and images from the HTML5 project into the `assets/www/` directory of the PhoneGap project (note that you do not need things unique to HTML5, such as the cache manifest)
- Make sure that your HTML entry point filename matches the path you used with the `loadUrl()` call in your activity (e.g., `index.html`)
- Add a reference to the PhoneGap JavaScript file from your HTML
- Build and install the project

Here is the `DroidGap` activity for our app, from the `PhoneGap/Checklist` project:

```
package com.commonsware.pg.checklist;

import android.app.Activity;
import android.os.Bundle;
import com.phonegap.DroidGap;

public class Checklist extends DroidGap {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    super.loadUrl("file:///android_asset/www/index.html");
  }
}
```

Here is the manifest, with all of the PhoneGap-requested settings added:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.pg.checklist"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:configChanges="orientation|keyboardHidden"
              android:label="@string/app_name"
              android:name="Checklist">
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
  <supports-screens android:anyDensity="true"
                    android:largeScreens="true"
                    android:normalScreens="true"
                    android:resizeable="true"
                    android:smallScreens="true" />
  <uses-permission android:name="android.permission.CAMERA" />
  <uses-permission android:name="android.permission.VIBRATE" />
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
  <uses-permission
android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />
  <uses-permission android:name="android.permission.READ_PHONE_STATE" />
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.RECEIVE_SMS" />
  <uses-permission android:name="android.permission.RECORD_AUDIO" />
  <uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS" />
  <uses-permission android:name="android.permission.READ_CONTACTS" />
  <uses-permission android:name="android.permission.WRITE_CONTACTS" />
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
</manifest>
```

And here is the HTML — almost identical to the HTML5 original, removing some HTML5 offline stuff (e.g., iOS icons) and adding in the reference to PhoneGap's JavaScript file:

```
<!DOCTYPE html>
<html lang="en" manifest="checklist.manifest">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Checklist</title>
    <meta name="viewport"
      content="width=device-width; initial-scale=1.0; maximum-scale=1.0;
user-scalable=0;" />
    <link rel="stylesheet" href="styles.css" />
    <script type="text/javascript" charset="utf-8"
src="phonegap.0.9.4.js"></script>
</head>
<body>
    <section>
        <header>
          <button type="button" id="sendmail">Mail</button>
          <h1>Checklist</h1>
        </header>
        <article>
          <form id="inputarea" onsubmit="addNewItem()">
              <input type="text" name="name" id="name" maxlength="75"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
                      autocorrect placeholder="Tap to enter a new item&hellip;"
/>
              <button type="button" id="add">Add</button>
            </form>
            <ul id="maillist">
            <li class="empty"><a href="" id="maillink">Mail remaining
items</a></li>
          </ul>
            <p id="totals"><span id="tally1">Total: <span
id="total">0</span></span>
            <span id="tally2">Remaining: <span
id="remaining">0</span></span></p>
          <ul id="checklist">
            <li class="empty">Loading&hellip;</li>
          </ul>
        </article>
        <fieldset>
          <button type="button" id="deletechecked">Delete Checked</button>
          <button type="button" id="deleteall">Delete All</button>
        </fieldset>
    </section>
    <script src="main.js"></script>
</body>
</html>
```

For many applications, this is all you will need — you are simply looking at PhoneGap to give you something you can distribute on the Play Store, on the iOS App Store, and so on.

Adding PhoneGap APIs

If you want to take advantage of more device capabilities, you can augment your HTML5 application to use PhoneGap-specific APIs. These run the gamut from telling you the device's model to letting you get compass readings. Hence, their complexity will vary. For the purposes of this chapter, we will look at some of the simpler ones.

Set up Device-Ready Event Handler

For various reasons, PhoneGap will not be ready to respond to all of its APIs right away when your page is loaded. Instead, there is a `deviceready` event that you will need to watch for in order to know when it is safe to use PhoneGap-specific JavaScript globals. The typical recipe is:

- Add an `onload` attribute to your `<body>` tag, referencing a global JavaScript function (e.g., `onLoad()`)

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- In `onLoad()`, use `addEventListener()` to register another global JavaScript function (e.g., `onDeviceReady()`) for the `deviceready` event
- In `onDeviceReady()`, start using the PhoneGap APIs

Use What PhoneGap Gives You

PhoneGap makes a number of methods available to you through a series of virtual JavaScript objects. Here, "virtual" means that you cannot check to see if the objects exist, but you can call methods and read properties on them. So, for example, there is a `device` object that has a handful of useful properties, such as `phonegap` to return the PhoneGap version and `version` to return the OS version. These virtual objects are ready for use in or after the `deviceready` event.

For example, here is a JavaScript file (`props.js` from the `PhoneGap/ChecklistEx` project) that implements an `onLoad()` function (to register for `deviceready`) and an `onDeviceReady()` function (to use the `device` object's properties):

```
// PhoneGap's APIs are not immediately ready, so set up an
// event handler to find out when they are ready

function onLoad() {
  document.addEventListener("deviceready", onDeviceReady, false);
}

// Now PhoneGap's APIs are ready

function onDeviceReady() {
  var element=document.getElementById('props');

  element.innerHTML='<li>Model: '+device.name+'</li>' +
                    '<li>OS and Version: '+device.platform +'
'+device.version+'</li>' +
                    '<li>PhoneGap Version: '+device.phonegap+'</li>';
}
```

The `onDeviceReady()` function needs a list element with an `id` of `props`.

The resulting app looks like:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

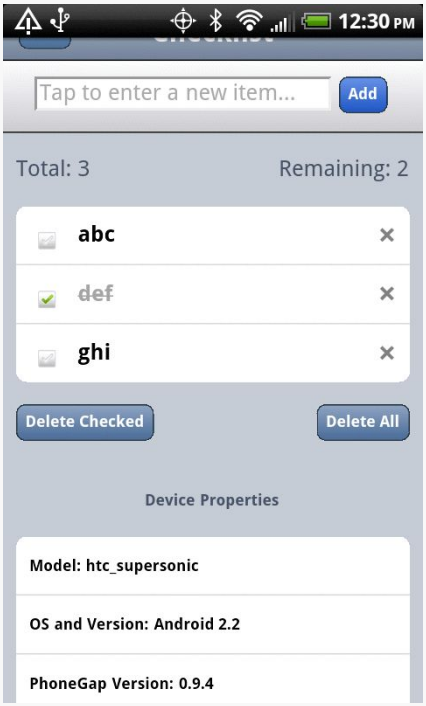


Figure 823: The PhoneGap Checklist application with device properties

Obviously, reading a handful of properties is far simpler than, say, taking a picture with the device's camera. However, the difference in complexity is mostly in what PhoneGap's virtual JavaScript objects give you and how you can use them, more so than anything peculiar to Android.

Issues You May Encounter

PhoneGap is a fine choice for creating cross-platform applications. However, it is not without its issues. Some of these issues may be resolved in time; some may be endemic to the nature of PhoneGap.

Security

Android applications use a permission system to request access to certain system features, such as making Internet requests or reading the user's contacts. Applications must request these permissions at install time, so the user can elect to abandon the installation if the requested permissions seem suspect.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A general rule of thumb is that you should request as few permissions as possible, and make sure that you can justify why you are requesting the remaining permissions.

PhoneGap, for a new project, requests quite a few permissions:

1. CAMERA
2. VIBRATE
3. ACCESS_COARSE_LOCATION
4. ACCESS_FINE_LOCATION
5. ACCESS_LOCATION_EXTRA_COMMANDS
6. READ_PHONE_STATE
7. INTERNET
8. RECEIVE_SMS
9. RECORD_AUDIO
10. MODIFY_AUDIO_SETTINGS
11. READ_CONTACTS
12. WRITE_CONTACTS
13. WRITE_EXTERNAL_STORAGE
14. ACCESS_NETWORK_STATE

Leaving this roster intact will give you an application that can use every API PhoneGap makes available to your JavaScript... and an application that will scare away many users. After all, it is unlikely that your application will be able to use, let alone justify, all of these permissions.

It is certainly possible for you to trim down this list, by modifying the `AndroidManifest.xml` file in the root of your PhoneGap project. However, you will then need to thoroughly test your application to make sure you did not get rid of a permission that you actually need. Also, it may be unclear to you which permissions you can safely remove.

Eventually, the PhoneGap project may have tools to help guide you in the choice of permissions, perhaps by statically analyzing your JavaScript code to see which PhoneGap APIs you are using. In the meantime, though, getting the proper set of permissions will involve a lot of trial and error.

Screen Sizes and Densities

Normal Web applications primarily focus on screen resolution and window sizes as their primary variables. However, mobile Web applications will not have to worry

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

about window sizes, as browsers and apps typically run full-screen. Mobile Web applications will need to deal with physical size and density, though — issues that are "off the radar" for traditional Web development.

Netbooks can have screens that are 10" or smaller. Desktops can have screens that are 24" or larger. On the surface, therefore, physical screen size would seem to be something Web developers would need to address. However, generally, screen resolution (in pixels) tracks well with physical size in the netbook/notebook/desktop realm. That is because screen density is fairly consistent across their LCDs, and that density is fairly low.

Smartphones, on the other hand, have several different densities, causing the connection between resolution and size to be broken. Some low-end phones, particularly with small (e.g., 3") LCDs, have densities on par with nice monitors. Mid-range phones have twice the density (240dpi versus 120dpi). Apple's iPhone 4 has even higher density, and there are some Android devices with similar densities. Hence, an 800x480 resolution could be on a screen ranging anywhere from 4" to 7", for example. Tablets add even more possible sizes to the mix.

This is compounded by the problems caused by touchscreens. A mouse can get pixel-level precision in its clicks. Fingers are much less precise. Hence, you tend to need to make your buttons and such that much bigger on a touchscreen, so it can be "finger-friendly".

This causes some problems with scaling of assets, particularly images. What might be "finger-friendly" on a low-density 3" device might be entirely too small for a high-density 4" device.

Native Android applications have built-in logic for dealing with this issue, in the form of multiple sets of "resources" (e.g., images) that can be swapped in based upon device characteristics. Eventually, PhoneGap and similar tools will need to provide relevant advice for their users for how to create applications that can similarly adapt to circumstances.

Look and Feel

A Web app never quite looks like a native one. This is not necessarily a bad thing. However, some users may find it disconcerting, particularly since they will not understand why their newly-installed app (made with PhoneGap, for example) would necessarily look substantially different than any other similar app they may already have.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

As HTML5 applications become more prominent on Android, this issue should decline in importance. However, it is something to keep in mind for the next year or two.

For More Information

At the moment, the best information on PhoneGap can be found on the PhoneGap site, including their API documentation.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Other Alternative Environments

The alternative application environments described in the preceding chapters are but the tip of the iceberg. Here, we will take a look at a few other alternative application environments, from the growing flood of such technologies.

Note that this area changes rapidly, and so the material in this chapter may be somewhat out of date relative to the progress each of these technologies has made.

Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate. Reading the introduction to this trail might not be a bad idea.

Rhodes

Spiritually, Rhodes is similar to PhoneGap, in that you develop an Android application whose user interface is defined via HTML, CSS, and JavaScript. The difference is that Rhodes bakes in a full Ruby environment, with a Rails-esque framework. Your Ruby code generates HTML and such to be "served" to an activity via a `WebView` widget, much like a server-side Ruby Web app would generate HTML to be served to a standalone Web browser.

Similar to PhoneGap, you can either build the project on your development machine or use their hosted build process. The latter is recommended, partly because the requirements for local builds are higher than those for PhoneGap — notably, Rhodes requires the Native Development Kit (NDK) for building and linking the Ruby interpreter to your application.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Rhodes winds up creating larger applications than does PhoneGap, due to the overhead of the Ruby interpreter (~1.5MB). However, if you are used to server-side Web development, Rhodes may be easier for you to pick up than would PhoneGap.

Flash, Flex, and AIR

Adobe has been hard at work extending their Flash, Flex, and AIR technologies to the mobile space. You can use Flex (the "Hero" edition) and Flash Builder (the "Burrito" edition... raising the question of whether the "hero" is hungry) to create Android APK files that can be distributed on the Play Store and deployed to Android devices. Those devices will need to have the AIR runtime installed — this is free, but a large download, and it only works on Android 2.2+ devices. The same projects can be repackaged for iOS and Blackberry OS 10 devices, and possibly future devices down the road.

AIR does not have quite as tight of integration to the platform as does PhoneGap (e.g., no access to the device's contacts), though one imagines that this is an area on which Adobe will devote more resources over time. And, Adobe is a large firm, with a large ecosystem behind it and many existing Flash, Flex, and AIR developer resources to tap into.

Note, though, that Adobe is officially discontinuing the Flash plug-in for Android after the Android 4.0 (Ice Cream Sandwich) release, which casts some doubt as to their long-term plans in the Flash/AIR space on mobile.

JRuby and Ruboto

One of the most popular languages designed to run on the JVM — besides Java itself — is JRuby. JRuby was quickly ported to run on Android, but with some optimizations disabled, since JRuby is really running on the Dalvik virtual machine that underlies the Android environment, not a classic Java VM.

However, JRuby alone cannot create Android applications. As a scripting language, there is no way for it to define an activity or other component — those need to be registered in the application's manifest as regular Java class files.

This is where Ruboto comes in.

Ruboto is a framework for a generic JRuby/Android application. It provides skeletal activities via a code generator and allows JRuby scripts to define handlers for all of

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

the lifecycle methods (e.g., `onCreate()`), plus define user interfaces using JRuby code, etc. The result can be packaged up as an APK file using supplied Rake script. The results can be uploaded to the Play Store or distributed however else you desire.

App Inventor

App Inventor is an Android application development tool originally made available by Google, but outside of the normal Android developer site. App Inventor was originally developed for use in education, but they have been inviting others into their closed beta.

App Inventor is theoretically a Web-based development tool. Here, "theoretically" means that, in practice, users have to do a fair amount of work outside of the browser to get everything set up:

1. Have Java installed and functioning in the browser, capable of running Java Web Start (.jnlp) applications
2. Download and install a large (~55MB) client-side set of tools
3. Have a phone and have it configured to work with App Inventor and the Android SDK

Once set up, App Inventor gives you a drag-and-drop GUI editor:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

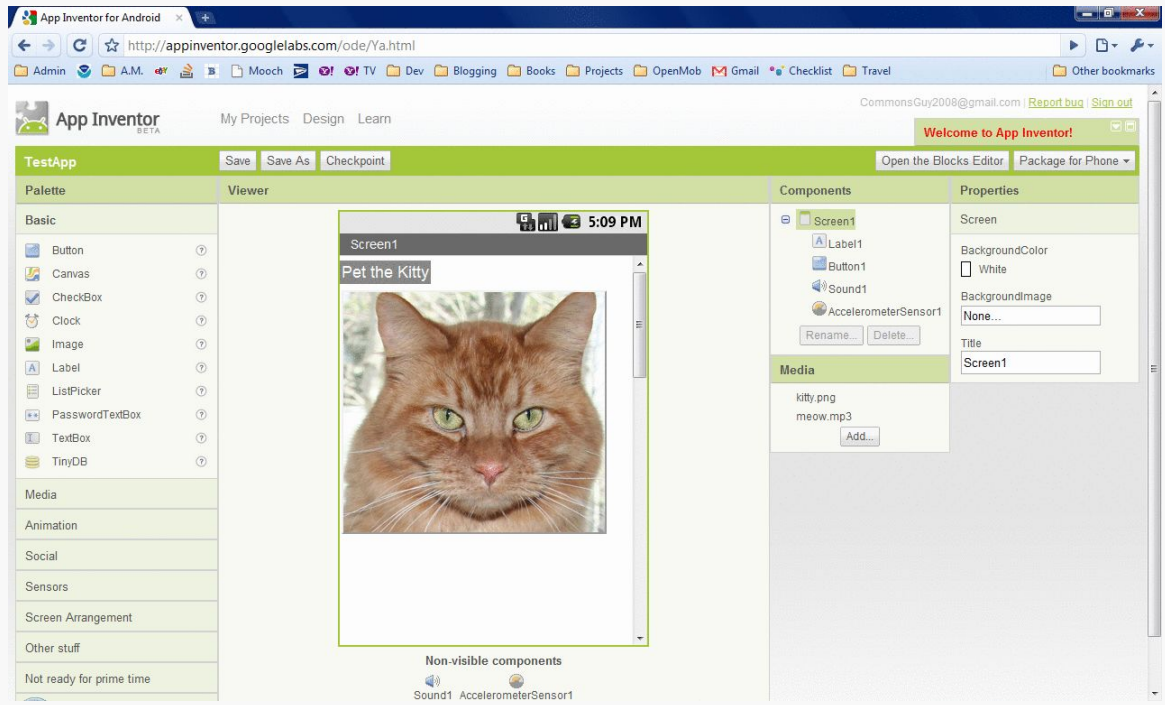


Figure 824: The App Inventor "Designer" view

... and a "blocks" editor, where you attach behaviors to events (e.g., button clicks) by snapping together various "blocks" representing events, methods, and properties:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

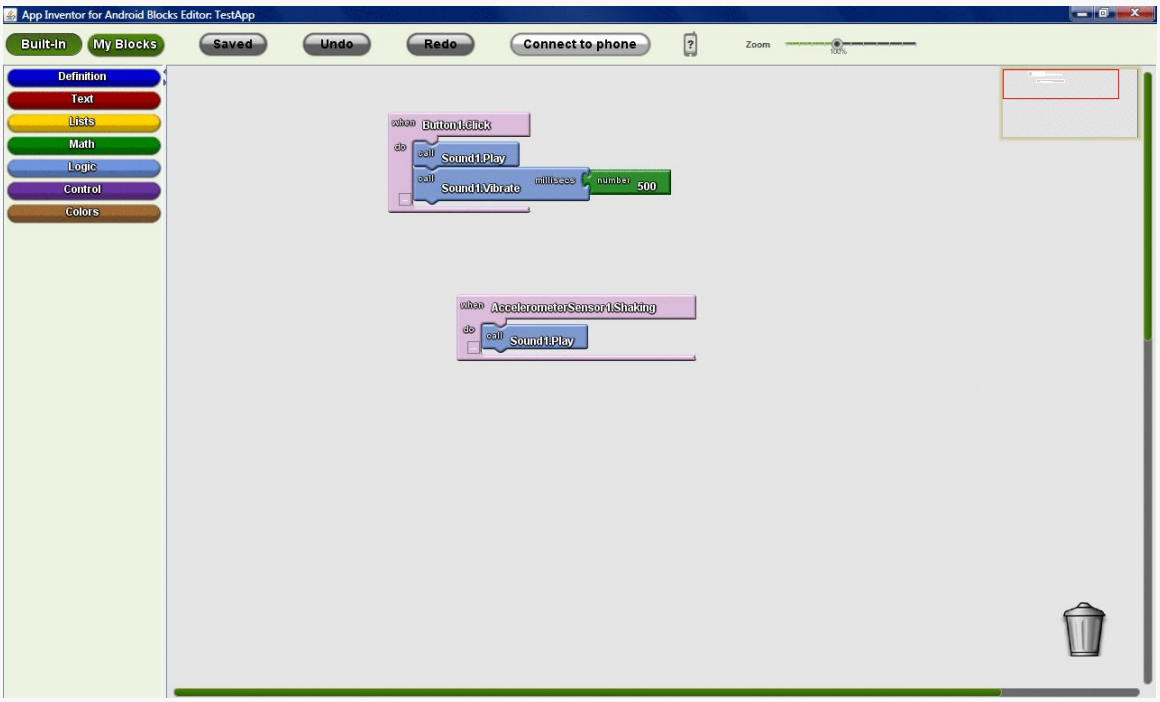


Figure 825: The App Inventor "Blocks" view

While working in the GUI editor, you see what you are building live on an attached phone and can be tested in real time. Later, when you are ready, you can package the application into a standard APK file.

However, App Inventor is not really set up for production application use today:

1. You cannot distribute App Inventor apps on the Play Store
2. It has more components aimed at "sizzle" (e.g., Twitter integration) and fewer delivering capabilities that a typical modern app might need (e.g., relational databases, lists)
3. Only one developer at a time can work on a project

In 2011, Google discontinued direct support for App Inventor, electing to transfer the project to MIT's Media Lab for ongoing development.

Titanium Mobile

Titanium Mobile's claim to fame is using JavaScript to completely define the user interface, eschewing HTML entirely. Rather, their JavaScript library — in addition to

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

providing access to databases and platform capabilities — also lets you declare user interface widgets. Its layout capabilities, for positioning said widgets, leaves a bit to be desired.

As of the time of this writing, Appcelerator — the creators of Titanium Mobile — does not offer a cloud-based set of tools. Their Titanium tool has a very slick-looking UI, but it still requires the Java SDK and Android SDK in order to be able to build Android applications, making the setup a bit daunting for some.

As of the time of this writing, Titanium Mobile supports development for Android and iOS, with Blackberry support in a private beta.

Other JVM Compiled Languages

If your issue is less with regular Android development, but you just do not like Java, any language that can generate compatible JVM bytecode should work with Android. You would have to modify the build chain for that other language to do the rest of the Android build process (e.g., generate `R.java` from the resources, create the APK file in the end).

Scala and Clojure are two such languages, for which their respective communities have put together instructions for using their languages for Android development.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Miscellaneous Topics

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Anti-Patterns

Much of this book has been focused on what you should do. In contrast, this chapter is focused on what you should *not* do.

All platforms have their anti-patterns: things that are technically possible but are not in the best interests of the users of that platform. Android is no exception. Some anti-patterns are simply annoying to users, while other anti-patterns can significantly infringe upon a user's use of their Android device, or even the user's freedom.

Much as the Hippocratic Oath directs doctors to "first, do no harm", Android application developers owe it to the users of their apps to avoid these anti-patterns to the greatest extent possible.

Prerequisites

This chapter assumes that you have read much of the book, particularly the core chapters.

Leak Threads… Or Things Attached to Threads

Leaking a thread means that you start a thread and never cause it to stop. For example, you might start a thread that runs in an infinite loop, doing some work and then sleeping for a while. The problem with infinite loops is that "infinite" is an awfully long time.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

All threads should clean up, in a timely fashion, when the component (e.g., activity, service) that started the thread is destroyed — or, in the case of an activity, perhaps just moved into the background.

How you ensure that the thread gets cleaned up is up to you. For threads doing transactional work, such as literally running a database transaction, it may be fine to just let them run to completion and shut down of their own accord. For "infinite" loops, there should be some way to tell the thread that it is no longer needed, such as via an `AtomicBoolean` flag, or using something more structured than a plain timing loop, such as a `ScheduledExecutorService`.

Also, bear in mind that you are responsible for threads that are created, on your behalf, by other things that you do. The most common leak scenario here comes with listeners associated with system services, like `LocationManager` and `SensorManager`. If you register a `LocationListener` via `requestLocationUpdates()` and fail to unregister that listener, you will not only be leaking the listener, but the component associated with that listener, and every system resource tied to that listener, such as any background threads.

The Costs

Threads are intrinsically static in scope. Hence, any object they can reach, directly or indirectly, cannot be garbage-collected while the thread is still running. Hence, if an activity forks a thread, it might do so using an anonymous inner class:

```
new Thread() {
  public void run() {
    // do something
  }
}).start();
```

Instances of an inner class — anonymous or otherwise — have an implicit reference back to the object that created them. Hence, the `Thread` would hold onto the `Activity` that created the thread, which in turn would hold onto all of its widgets and so forth. None of that can be garbage-collected until after the thread terminates, even if the activity is destroyed.

The Counter-Arguments

I want the thread to keep running even after the activity is destroyed

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In this case, the thread should be created and managed by a service, not simply leaked. Not only does this give you an opportunity to clean up the thread when needed, but it also alerts Android that you are still trying to do some work, so Android will not necessarily terminate your process very quickly.

However, be careful about assuming that you can have a thread — even one managed by a service — run forever, as you will see in the next couple of sections.

I do not know when the thread is no longer needed

Then you have a serious design problem.

A common variation on this theme is:

The thread is needed so long as I have an activity in the foreground

This is a bit tricky, as Android does not really expose the concept of *applications* being in the foreground, just activities.

The safest course of action is to have the thread be managed by a service, then keep track of whether or not you have an activity in the foreground. For example, in `onPause()` of each activity, use `postDelayed()` to return control to you after a short delay, and in `onResume()`, update a timestamp of your last return to the foreground (held in a static data member). When the `Runnable` for `postDelayed()` executes, check that timestamp — if it is too old, you know that none of your activities are in the foreground, and you can stop the service, having it stop your thread.

Use Large Heap Unnecessarily

Encountering an `OutOfMemoryError` certainly sucks. These are caused either by a memory leak or by trying to use more memory than is practical given the device. For example, loading up lots of bitmaps can easily chew up your available heap space.

To some, therefore, `android:largeHeap` seems to be the perfect solution.

Added in API Level 11, `android:largeHeap` tells Android to give you a much larger heap size than is normally given to a process. So, instead of having 32MB or 48MB or so of heap, you might have 256MB of heap.

The right solution, in most cases, is to fix the underlying memory problem, not to mask it by requesting an over-sized heap.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Costs

To you, having hundreds of megabytes of extra heap may be a blessing. To the user, it may be a curse. That memory has to come from somewhere, and the "somewhere" is from other processes. Your app will force other apps' processes to be terminated far more quickly than normal, which may slow the user down when she tries to switch between your app and others. Your app may even materially harm the functionality of other apps, who have their processes terminated before they can finish their work, just to satisfy your memory craving.

Bear in mind that Android does not employ swap space (the Linux equivalent of a Windows pagefile). Hence, whereas Windows can allocate lots of memory and slows down as it goes, Android is far more limited, in accordance with its mobile roots.

Furthermore, in many cases, adding more heap space does not eliminate the problem, any more than spraying air freshener gets rid of the dead cat in your living room that is causing the odor. With a memory leak, for example, all the larger heap does is increase the time before you eventually run out of memory.

The Counter-Arguments

I really need to be able to manipulate large chunks of memory

There are certainly apps for which `android:largeHeap` is justified, such as complex data editors, such as image editors, video editors, etc.

Hence, in practice, the real anti-pattern is not using `android:largeHeap`, but rather in doing so for apps where the user would not feel that the resulting effects are justified. For example, neither a Twitter client, nor a banking app, should need a large heap, even if the developer is running into memory management issues.

Android makes it too hard to manage memory, so I need a large heap

There is no question that developing mobile applications is challenging, particularly when it comes to memory management. That is not unique to Android — embedded systems developers are used to writing apps where the heap size is better measured in KB instead of MB, for example.

Outside of bitmaps and massive data sets, though, it is a bit difficult to actually run out of memory. While a `TextView` may take up 1KB of heap space, it takes a *lot* of `TextView` widgets to chew through a 48MB heap.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The reason why bitmaps tend to trip up developers is that Android makes using them *too easy*. For example, it is simple to set a bitmap as a background of some container like a `LinearLayout`, where developers then blindly ignore the fact that if the bitmap is not *precisely* the size of the container, Android will need to scale the image, consuming more heap space.

Misuse the MENU Button

The MENU button on Android devices is designed to display either the options menu (on Android 1.x/2.x devices that are not using an action bar backport like ActionBarSherlock) or the action bar overflow menu.

The MENU button is *not* designed for any other purpose. Some developers have taken to using it for arbitrary aims, and that is a mistake.

The Costs

The MENU button does not exist on many Android devices. In particular, devices designed for Android 3.0 and higher do not *need* a MENU button. Some will have them, but most will not. Hence, anything that requires the MENU button will simply be unavailable on those devices.

And, as of Android 4.4, Google is putting increasing pressure on device manufacturers to dump the MENU button, making it less likely to appear in the future.

The Counter-Arguments

Well, if I keep `targetSdkVersion` below 11, I can have a soft MENU button

This is true, insofar as a menu affordance will be added to the system bar or navigation bar on devices that lack a dedicated MENU button.

Whether the *user* is expecting to use this button is another thing entirely.

As more and more users run Android 3.0+ devices, they will use more and more apps that have `android:targetSdkVersion` set to 11 or higher. The remaining handful of apps that do not will be "weird". In particular, they may not notice the menu affordance, as they are not looking for one, or they may not know what it does, as they are not used to needing it.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Moreover, eventually, other things will drive you to want an `android:targetSdkVersion` higher than 10, as the menu affordance is not the only feature driven by this value. The sooner you can remove your dependence on a menu affordance, the sooner you can upgrade your `android:targetSdkVersion` to solve other problems that you are encountering.

I think the action bar is ugly, a waste of space, or otherwise bad

That's nice. It does not mean that you need a menu affordance and a tie to a MENU button.

For example, well-written games will have a menu integrated into the game UI itself. This was often done even before Android 3.0, since the options menu UI would not look much like the game's UI, and the developer wanted a consistent look-and-feel.

So long as the user recognizes how to reach the menu (e.g., a three-dots or three-bars icon), the menu does not have to be driven by Android, but instead could be handled by your app directly. You can see this in the Google Navigation app, which avoids an action bar but still displays its own menu from its own on-screen menu affordance.

Interfere with Navigation

Some developers try to take over the device. They attempt to block the use of anything not related to their app: the HOME key, the recent tasks list, the notification drawer, etc.

Android treats such behavior as malware. Android is designed to keep control of the device in the hands of the user and tries very hard to prevent apps from stealing that control.

The Costs

While there are certain cases where blocking navigation outside the app may seem justified (see the counter-arguments, below), there is simply too much opportunity for malfeasance. Users tend to want to use their devices on their terms, not necessarily the terms of some random developer. Malware authors, in particular, love to learn about script-kiddie hacks that allow them to control a device, and by extension, control the users.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The Counter-Arguments

I am writing a lock screen

No, you are not. You are writing something that you *think* is a lock screen. Really what you are writing is something that weakens device security... if the app in question is designed to be downloaded and run on arbitrary devices.

Android devices can be rebooted into "safe mode". Much like the Windows boot option that bears the same name, "safe mode" only runs apps that are part of the system firmware, not any third-party apps.

So, let's assume that the user installs your "lock screen". Inevitably, part of the setup of a third-party "lock screen" is to disable any sort of security that is part of the native lock screen, so the user does not have to unlock things twice. Even though your lock screen may implement all sorts of security, all somebody else has to do is reboot the device in safe mode, and they now have complete access to the device, including the ability to uninstall your lock screen. By contrast, the native lock screen is in force even if the device reboots in safe mode.

I am writing a parental control app

Rebooting in safe mode is within the motor-control skills of your average three-year-old child. Hence, the primary limitation is whether or not the child knows *how* to reboot the device in safe mode, which they can learn from the Internet, friends, etc. And, if the device is really an adult's device, where the "lock screen" allows access to a subset of child-friendly apps, the real risk is not from the child rebooting the device in safe mode, but from the crook who steals the device rebooting in safe mode.

I am writing a lock screen designed to run on whole-disk-encrypted devices

While whole disk encryption — available on Android 4.0+ — does solve the issue of rebooting in safe mode, bear in mind that users then cannot disable the required password security on the native lock screen, as that is tied into the whole disk encryption process.

I am writing a kiosk app

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Here, the term "kiosk app" refers to an app that represents the functionality of a single-purpose device. For example, a restaurant might want to distribute menus to customers in the form of a tablet app; the menu app would be the "kiosk app".

In this case, the owner of the device is the one trying to lock it down to be single-purpose. That is completely reasonable... except that it runs counter to the behavior of standard consumer builds of Android.

The right solution, in this case, is to create custom firmware for the single-purpose devices. This firmware can set up the kiosk app to be the home screen (thereby blunting the effectiveness of HOME, BACK, etc.), and modifications to the firmware can apply access controls to other aspects of the device (e.g., notifications). Unfortunately, there are few (if any) businesses set up to help create such single-purpose firmware for single-purpose devices.

Use android:sharedUserId

If you are creating more than one application, where those applications should be sharing data, you may be tempted to use `android:sharedUserId`. This attribute, applied to the root `<manifest>` element in your manifest, allows two or more apps to share a Linux user account. That will allow these apps full access to the other apps' files. The limitations are that you must use the same value for `sharedUserId` and that all such apps must be signed with the same signing key.

However, this is a fairly crude and somewhat risky approach to sharing information between apps. In most cases, you will be better served using any of the structured IPC options within Android, such as remote services and content providers.

The Costs

First, you must make the decision to use `android:sharedUserId` before you ever ship your app in production. Should you change the `sharedUserId` value — or switch from no value to a new value — when your change is installed, the new version of your app will have no rights to access the old version of your app's files. This is unlikely to turn out well.

Second, it will be up to you to maintain data integrity of these files in the face of simultaneous access from multiple apps. SQLite should handle this for you for your databases, as it is set up to use process-level locking — this is why SQLite can be used as the out-of-the-box database solution for Web frameworks like Rails.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, any other sort of file, including `SharedPreferences`, will lack that coordination, unless you somehow arrange to do it yourself. And even the SQLite-level coordination has its limits, as one app has no way to know about another app's changes to the data, except by re-querying the database.

Third, using `android:sharedUserId` limits your flexibility. You cannot use it with third-party apps. You cannot readily sell one of your apps in your suite, as then it becomes a third-party app and can no longer be signed by the same signing key as are the rest of your apps. Basically, `sharedUserId` causes multiple separate APKs to behave, in some respects, as one larger APK.

The Counter-Arguments

I need the ensure only my apps can share the data, not others

Use a signature-level permission. This gives you the same level of security as does `android:sharedUserId` without most of the risks.

Writing IPC code is tedious

So is writing cross-process data integrity code.

Implement a "Quit" Button

Perhaps the most contentious question and answer on Stack Overflow's `android` tag is "Quitting an application - is that frowned upon?". This exchange is nearly three years old (as of the time of this writing), yet the answer receives both upvotes (and a few downvotes) with some regularity.

Other Android experts, such as Reto Meier, have weighed in on the issue and have offered similar recommendations – that is, do not have a "quit" or "exit" button in your app.

(here, "button" is shorthand for any command-style interface, and includes menu options, action bar items, and the like by extension)

The reason is simple: whatever your "quit" or "exit" button does *should be happening in other conditions as well*, and handling those other conditions should eliminate the need for the button.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If the app moves into the background *for any reason*, you need to treat the user and her device with respect. This means stopping background threads that are not needed, releasing system resources like the GPS radio (immediately or after a modest delay), and the like. The user should not need to "quit" your app to accomplish this, because your app will move to the background for other reasons, such as incoming phone calls, or the user pressing the HOME button.

The Costs

You might think "well, what's the harm in having the 'quit' button that, say, just calls `finish()`?"

First, rarely is it that simple. Calling `finish()` will return the user to the previous activity, and so for any multi-activity app, there will be scenarios where `finish()` is not really "quit". The only simple thing you can universally do is have "quit" bring up the home screen, in which case all you have done is waste screen real estate duplicating the HOME button functionality. Worse, the developer might say "oh, well, I will just terminate my process when they press 'quit'", and *that* anti-pattern is coming up next in this chapter.

Second, the user will start to think that they *need* to press "quit", or else bad things might happen. They will see an explicit "quit" option and start to wonder "well, gee, when am I supposed to press that, and what happens if I do not?" This, in turn, will lead to the user going out of their way to make sure to press your "quit" button, even if doing so does not actually change anything about the behavior of your app, courtesy of the placebo effect.

The Counter-Arguments

I need to let the user log out of the app, so I need a "quit" button

No, you need a "logout" button that clears your cached authentication credentials (e.g., sets a static data member to `null`), then brings up the login activity using `FLAG_ACTIVITY_SINGLE_TOP` and `FLAG_ACTIVITY_CLEAR_TOP` to wipe out all other activities in your process. And, probably, you need to have some sort of inactivity-based "timeout" that also logs out the user (e.g., sets that static data member to `null`).

I am running stuff in the background, so I need a "quit" button

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

No, you need a "stop that background stuff" button, preferably with a shorter, more specific label. And, you need that to also be available from the `Notification` that you are using with your foreground service, where applicable.

Terminate Your Process

Closely related to the above anti-pattern is to forcibly stop your process, such as via `System.exit()`, `Runtime.exit()`, `Process#killProcess()`, and so forth. These are often used in concert with an in-app "quit" button, or sometimes for other reasons (e.g., could not figure out how to handle an exception gracefully).

The Costs

Simply put, Google has warned, repeatedly, that there may be side effects from terminating your own process, rather than having Android do proper cleanup first.

- "You should really think about not exiting the application. This is not how Android apps usually work." (Romain Guy)
- "To be clear: using System.exit() is strongly recommended against, and can cause some poor interactions with the system. Please don't design your app to need it." (Dianne Hackborn)
- "There is no reason or need to call [exit()]" (Dianne Hackborn)
- "Nobody has said anything about Process.kill() not doing anything. You want to kill your *own* process and cause the user to experience your *own* application having weird behavior at times due to it? Have at it. I just want to be clear that this is not what we recommend doing... and you are likely to cause bad behavior in your app at least at times due to it... There is no API to quit an application, because there is no such concept on Android, and trying to implement such a thing is going to result in fighting against how Android works." (Dianne Hackborn)

The Counter-Arguments

I am using a C library that is buggy, so I need to terminate my process

Fix the bugs in the library. For example, C libraries that rely too heavily on global variables may need to be adjusted to use session handles that get passed around.

Well, it is not my C library, but one from a third party, so I need to terminate my process

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Find a library that is Android-compatible, then. It is likely that you will encounter other problems with this library, if it is not designed to work on Android (e.g., not set up to work properly on ARM CPUs).

There is a bug in Android for which I have found no workaround short of terminating my process

This is one of the few legitimate reasons for terminating a process, but it is so rare that it is difficult to find a citation of a place where such a bug (and workaround) exists.

I need to do *something* from my top-level exception handler!

Set relevant static data members to `null`, then start up your launcher activity, using `FLAG_ACTIVITY_SINGLE_TOP` and `FLAG_ACTIVITY_CLEAR_TOP` to wipe out all other activities in your task. This should reset you to your original state, as if the user had launched the app.

Try to Hide from the User

Some developers view the user as the enemy. These developers try to insulate their app from the user, to make data inaccessible to the user, to make the app "unkillable" by the user, etc. In many cases, this is at the behest of some enterprise, wanting to exert control over the user's use of the app or even the device.

Android is a consumer operating system. It is designed to put power in the hands of whoever is holding the device and can authenticate themselves to the device (e.g., via a password on the lock screen). Enterprises and malware authors have much the same interests: they wish to take control away from the user and give the control to somebody else. Android defends against malware; enterprises get caught in the crossfire.

Inevitably, the right solution here will be an enterprise remix of Android, designed to be loaded on enterprise-supplied devices, that put the control in the hands of the enterprise.

The Costs

Simply put, you are wasting your time, which could be better spent on other pursuits.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

With respect to data, if your app can access that data, by definition, a sufficiently talented user can get at the data:

- If you put it on internal storage, the user can root the device
- If you further encrypt the data, the user can find the encryption algorithm and key in your app, then decrypt the data
- If you try obfuscation or other techniques to mask the encryption algorithm and key, the user will use cracking tools to find this information anyway, or will transfer your app to a ROM mod that contains a modified version of the Android framework that can collect this information when you go to decrypt the data
- And so on

With respect to the process, the user can force-stop anything installed app via the Settings app. And, even if you use script-kiddie tricks to try to prevent access to Settings, the user can nuke your app from orbit via the command line, using the full Android SDK or third-party tools.

The Counter-Arguments

I am creating an app for an enterprise, and we need to control the app

Then you further need to control the device, which leads to the "enterprise flavor of Android" solution mentioned earlier in this section.

I am creating a lock screen/parental control app/kiosk app

Please see the counter-arguments for "Interfering with Navigation" from earlier in this chapter.

Use Multiple Processes

Some Android professionals recommend the use of `android:process` to have components run in separate processes from the main one for an application. For example, you might have all of your activities in the main process but isolate a service in a separate process. Or, you might have some memory-intensive activity (e.g., an image editor) run in a separate process.

As with most of these anti-patterns, while the `android:process` feature is valid, it is rarely necessary. To some extent, developers get caught up in process isolation from

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

its use on servers and forget that mobile devices typically have fewer resources — RAM and CPU — than do their server counterparts. Few of Google's apps use `android:process`; even complex apps like Gmail or the original Browser avoid it.

The Costs

Each process gets its own heap space, cutting into the heap available for other applications. As with the large-heap anti-pattern discussed above, this will tend to force other apps to be ejected from memory sooner than normal, with commensurate impacts on user experience.

Inter-process communication (IPC) is not cheap, compared with normal method invocation within a process. Hence, tightly-coupled processes will chew through more CPU than their single-process counterparts. While it is unlikely that you will see major performance implications (unless you are doing a preposterous amount of IPC), this will consume more battery than is otherwise warranted.

The Counter-Arguments

I am using a C library that is buggy, and you told me not to terminate my process

As noted earlier, fix the bugs in the library.

Hello? It is not my C library, but one from a third party!

Find a library that is Android-compatible, then.

I need more heap space

On Android 3.0 and higher, `android:largeHeap` is available, though its misuse is another anti-pattern, discussed above. However, prior to Android 3.0, `android:largeHeap` was not an option. One workaround used by some apps is to fork several processes, thereby getting several "small" heap allocations (e.g., 32MB) instead of just one.

In cases where `android:largeHeap` is indeed justified, using multiple processes as a workaround on older Android versions is justified as well. However, bear in mind that IPC overhead is non-trivial, so have a plan to dump the multiple processes and use `android:largeHeap` once you drop support for Android 1.x/2.x.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

I want my UI not to freeze when doing background work

Use threads, not processes, for this.

Hog System Resources

Some of these anti-patterns, like the multiple-process one just now, are really concrete sub-types of a more general anti-pattern: assuming yours is the only app running on the device. While your app may be the only one running in the foreground (assuming that you actually *are* in the foreground), there are other apps in the background, and ones that soon will come to the foreground. You need to "play nice" and ensure that these other apps will have their fair share of system resources.

One example is open files on external storage. For some devices — but not all – there is a limit of 1,024 simultaneously open files. In principle, that should be plenty. However, if some app — maybe yours? — opens a whole bunch of files, it is possible that other apps trying to access external storage at that point will crash because the limit was hit.

The Counter-Arguments

Um, well, I'm just more important than those other developers

::facepalm::

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Widget Catalog

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Widget Catalog: AdapterViewFlipper

A regular `ViewFlipper` shows only one child widget or container at a time. So does an `AdapterViewFlipper`. The difference is where the children come from. With a regular `ViewFlipper`, you add children much like you would any other standard container class, such as defining the children in your layout XML resource. With `AdapterViewFlipper`, the children come from an `Adapter`.

While `AdapterViewFlipper` does not inherit from `ViewFlipper` (or vice versa, for that matter), their public API is largely the same:

- You can control which child is visible, either by index or via `showNext()`/`showPrevious()` methods to rotate between them.
- You can set up animated effects to control how a child leaves and the next one enters, such as applying a sliding effect.
- You can set up `AdapterViewFlipper` to automatically flip between children on a specified period.

There are two key advantages for `AdapterViewFliper`:

1. Since it uses an `Adapter` model, it can be more memory efficient for lots of children, through child view recycling
2. It is available for use in an app widget

However, `AdapterViewFlipper` is new to API Level 11 and is unavailable on older versions of Android. It is not included in the Android Support package backport.

Key Usage Tips

All of the usage tips from `ViewFlipper` are relevant for `AdapterViewFlipper`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A Sample Usage

The sample project can be found in WidgetCatalog/AdapterViewFlipper.

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<AdapterViewFlipper xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/details"
  android:layout_width="match_parent"
  android:layout_height="match_parent"/>
```

Activity:

```
package com.commonsware.android.avflip;

import android.app.Activity;
import android.os.Bundle;
import android.widget.AdapterViewFlipper;
import android.widget.ArrayAdapter;

public class FlipperDemo2 extends Activity {
  static String[] items= { "lorem", "ipsum", "dolor", "sit", "amet",
      "consectetuer", "adipiscing", "elit", "morbi", "vel", "ligula",
      "vitae", "arcu", "aliquet", "mollis", "etiam", "vel", "erat",
      "placerat", "ante", "porttitor", "sodales", "pellentesque",
      "augue", "purus" };
  AdapterViewFlipper flipper;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    flipper=(AdapterViewFlipper)findViewById(R.id.details);
    flipper.setAdapter(new ArrayAdapter<String>(this, R.layout.big_button,
items));
    flipper.setFlipInterval(2000);
    flipper.startFlipping();
  }
}
```

Visual Representation

There is no visual representation of an `AdapterViewFlipper` itself, as it renders no pixels on its own. Rather, it simply shows the current child.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Widget Catalog: CalendarView

`CalendarView`, as you might have guessed, displays a calendar to the user, designed to allow the user to pick a date. You supply a starting date, which the user then manipulates, triggering event listeners whenever the date is changed.

Note that this is a small calendar – it is not designed to show details *within* a date, such as appointments and times.

This view is available standalone and also as an optional adjunct to the `DatePicker` widget.

This view was added in API Level 11 and therefore will not be available on older versions of Android, though a backport is available that works on Android 2.2 onwards.

Key Usage Tips

If you do nothing, the `CalendarView` will start with today's date, though you can call a `setDate()` method to pass in a `Calendar` object to use to change the initially-selected date. You can also call `setOnDateChangeListener()` to supply an `OnDateChangeListener` to learn when the user changes the date in the `CalendarView`.

`CalendarView` works well with `Calendar` and `GregorianCalendar`, in terms of setting and getting the year/month/day-of-month from the `CalendarView` (as supplied to the `onSelectedDayChange()` method of your `OnDateChangeListener`) and converting it into something you can use in your code.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A Sample Usage

The sample project can be found in WidgetCatalog/CalendarView.

Layout:

```
<CalendarView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/calendar"
  android:layout_width="match_parent"
  android:layout_height="match_parent"/>
```

Activity:

```
package com.commonsware.android.wc.calendar;

import android.app.Activity;
import android.os.Bundle;
import android.widget.CalendarView;
import android.widget.CalendarView.OnDateChangeListener;
import android.widget.Toast;
import java.util.Calendar;
import java.util.GregorianCalendar;

public class CalendarDemoActivity extends Activity implements
    OnDateChangeListener {
  CalendarView calendar=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    calendar=(CalendarView)findViewById(R.id.calendar);
    calendar.setOnDateChangeListener(this);
  }

  @Override
  public void onSelectedDayChange(CalendarView view, int year,
                                  int monthOfYear, int dayOfMonth) {
    Calendar then=new GregorianCalendar(year, monthOfYear, dayOfMonth);

    Toast.makeText(this, then.getTime().toString(), Toast.LENGTH_LONG)
         .show();
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Visual Representation

This is what a `CalendarView` looks like in a few different Android versions and configurations, based upon the sample app shown above.

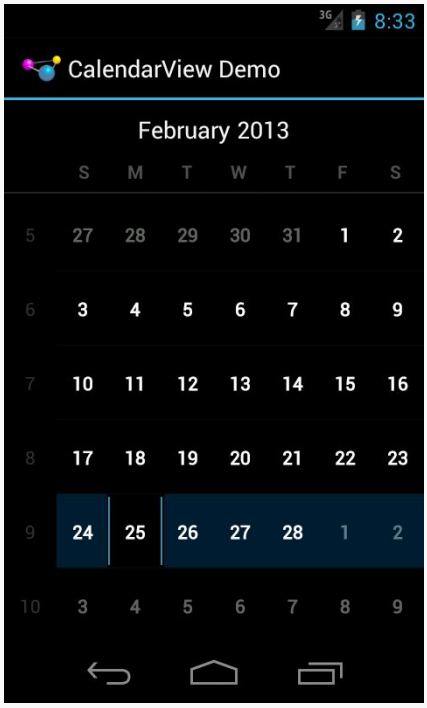


Figure 826: Android 4.0

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

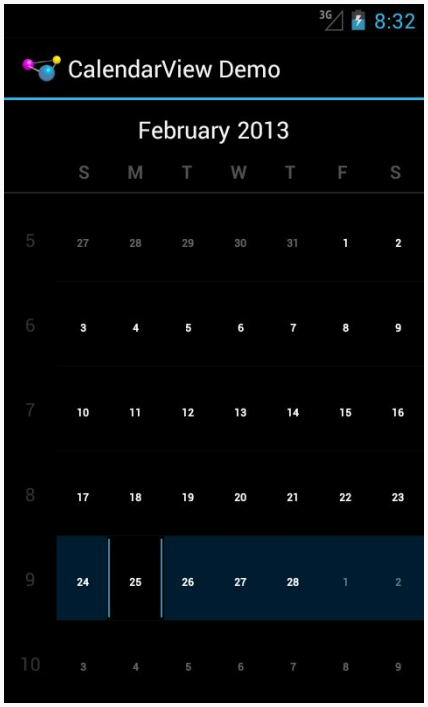
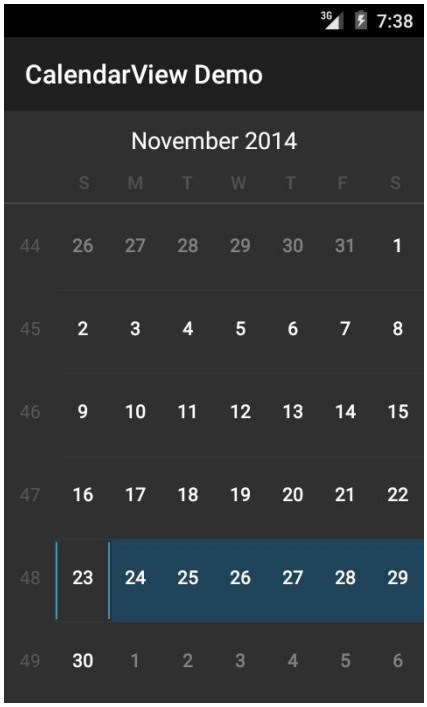


Figure 827: Android 4.1



Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Widget Catalog: DatePicker

`DatePicker`, as the name might suggest, allows the user to pick a date. You supply a starting date, which the user then manipulates, triggering event listeners whenever the date is changed.

Key Usage Tips

If you do nothing, the `DatePicker` will start with today's date. However, if you want to set up an `OnDateSetListener` to find out when the date changes, you will need to call `init()` to do so, in which you also need to set the date.

`DatePicker` works well with `Calendar` and `GregorianCalendar`, in terms of setting and getting the year/month/day-of-month from the `DatePicker` and converting it into something you can use in your code.

API Level 11 introduced an optional `CalendarView` adjunct to the `DatePicker`, determined via `setCalendarViewShown()` or `android:calendarViewShown`. This works well on `-normal` screens in landscape and on `-large`/`-xlarge` screens. On `-normal` screens in portrait, the year portion of the picker may be chopped off to save room. Using the `CalendarView` option on `-small` screens is probably not a good idea.

However, on Android 5.0, the `CalendarView` is *always* shown and cannot be removed, as the "picker" itself does not allow the user to pick a date. The user uses the `CalendarView` to pick a date, or taps on the year in the "picker" to choose a year. This means that `DatePicker` is not a particularly good widget to use, especially on smaller screens.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

A Sample Usage

The sample project can be found in WidgetCatalog/DatePicker.

Layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:gravity="center_horizontal">

  <DatePicker
    android:id="@+id/picker"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
    android:calendarViewShown="true"/>

  <CheckBox
    android:id="@+id/showCalendar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:checked="true"
    android:text="@string/calendar"/>

</LinearLayout>
```

Activity:

```
package com.commonsware.android.wc.datepick;

import android.app.Activity;
import android.os.Build;
import android.os.Bundle;
import android.view.View;
import android.widget.CheckBox;
import android.widget.CompoundButton;
import android.widget.CompoundButton.OnCheckedChangeListener;
import android.widget.DatePicker;
import android.widget.DatePicker.OnDateChangedListener;
import android.widget.Toast;
import java.util.Calendar;
import java.util.GregorianCalendar;

public class DatePickerDemoActivity extends Activity implements
    OnCheckedChangeListener, OnDateChangedListener {
  DatePicker picker=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    CheckBox cb=(CheckBox)findViewById(R.id.showCalendar);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.HONEYCOMB) {
      cb.setOnCheckedChangeListener(this);
    }
    else {
      cb.setVisibility(View.GONE);
    }

    GregorianCalendar now=new GregorianCalendar();

    picker=(DatePicker)findViewById(R.id.picker);
    picker.init(now.get(Calendar.YEAR), now.get(Calendar.MONTH),
                now.get(Calendar.DAY_OF_MONTH), this);
  }

  @Override
  public void onCheckedChanged(CompoundButton buttonView,
                               boolean isChecked) {
    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.HONEYCOMB) {
      picker.setCalendarViewShown(isChecked);
    }
  }

  @Override
  public void onDateChanged(DatePicker view, int year, int monthOfYear,
                            int dayOfMonth) {
    Calendar then=new GregorianCalendar(year, monthOfYear, dayOfMonth);

    Toast.makeText(this, then.getTime().toString(), Toast.LENGTH_LONG)
         .show();
  }
}
```

The `CheckBox` is tied to the visibility of the `CalendarView`. Since this is only available on API Level 11 and higher, we simply remove the `CheckBox` on earlier versions of Android, so we do not have to worry about whether or not the `CheckBox` gets unchecked by the user.

Visual Representation

This is what a `DatePicker` looks like in a few different Android versions and configurations, based upon the sample app shown above.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

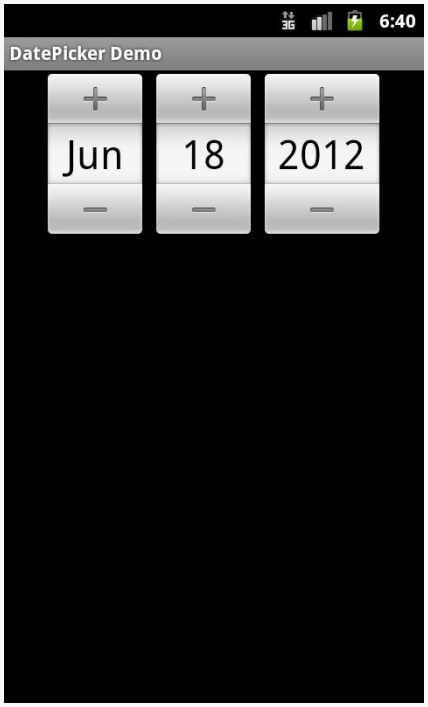


Figure 828: Android 2.3.3

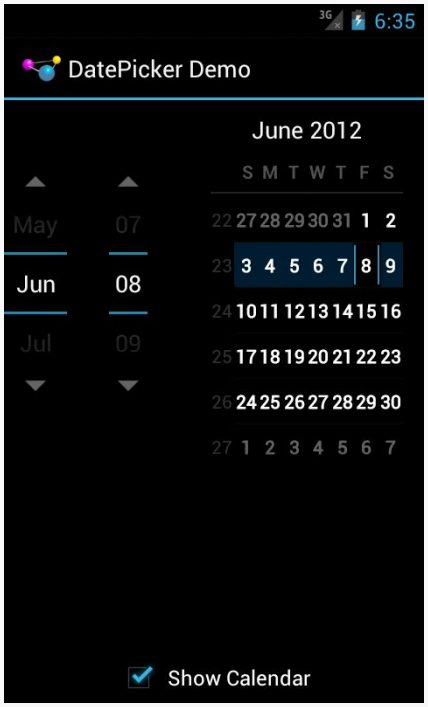


Figure 829: Android 4.0.3, with CalendarView, Portrait

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

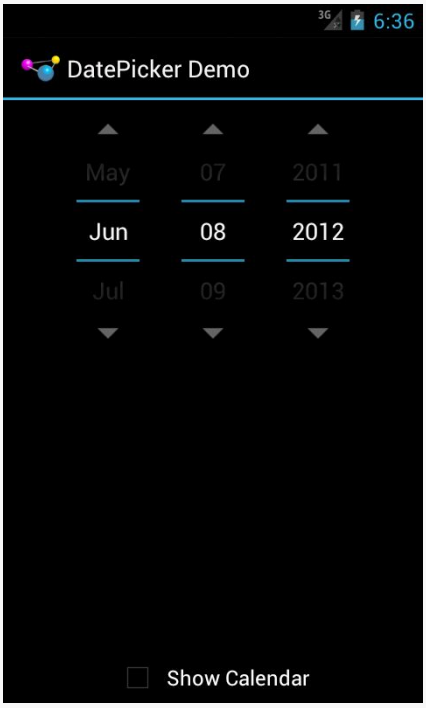


Figure 830: Android 4.0.3, without CalendarView, Portrait

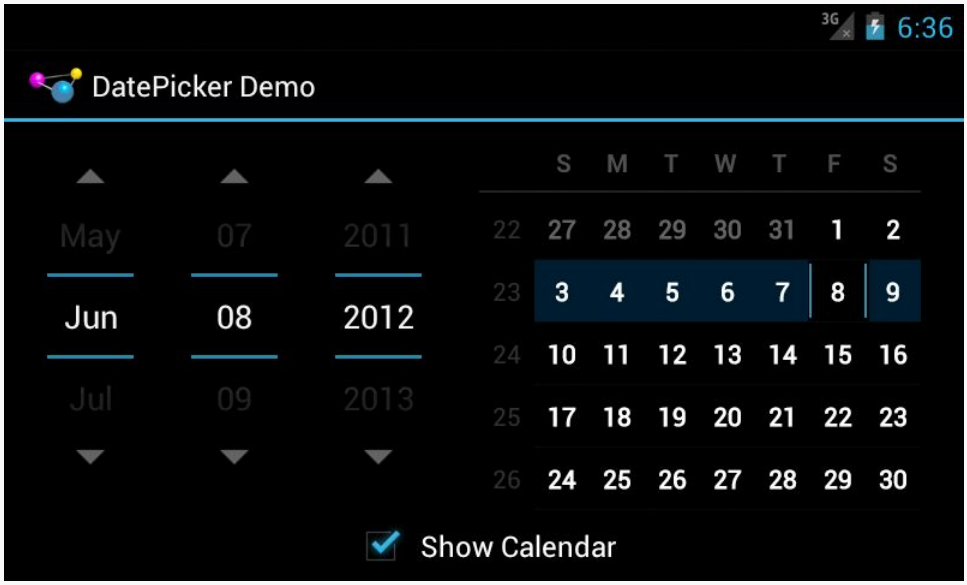


Figure 831: Android 4.0.3, with CalendarView, Landscape

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

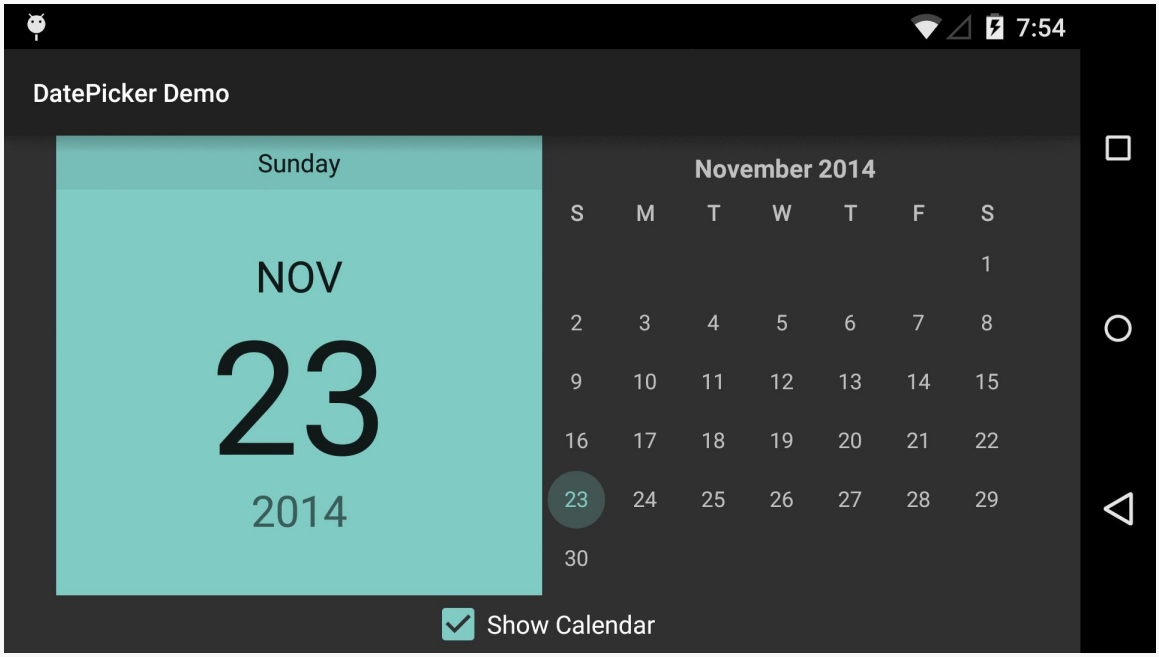


Figure 832: Android 5.0, with CalendarView, Landscape

—

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Widget Catalog: ExpandableListView

Android does not have a "tree" widget, allowing users to navigate an arbitrary hierarchy of stuff. In large part, that is because such trees are difficult to navigate on small touchscreens with comparatively large fingers.

Android *does* have `ExpandableListView`, a subclass of `ListView` that supports a two-layer hierarchy: groups and children. Groups can be expanded to show their children or collapsed to hide them, and you can get control on various events for the groups or the children.

Key Usage Tips

Android offers an `ExpandableListActivity` as a counterpart to its `ListActivity`. However, it does not offer an `ExpandableListFragment`. This is not a major issue, as you can work with an `ExpandableListView` inside a regular `Fragment` yourself, just as you would for most other widgets not named `ListView`.

Rather than use a `ListAdapter` with `ExpandableListView`, you will use an `ExpandbleListAdapter`, where you can control separate details for groups and children. These include:

- `SimpleExpandableListAdapter`, roughly analogous to `ArrayAdapter`, where your data resides in a `List` of `Map` objects for groups, and a `List` of a `List` of `Map` objects for the children
- `CursorTreeAdapter` and `SimpleCursorTreeAdapter`, roughly analogous to `CursorAdapter` and `SimpleCursorAdapter`, for mapping data in a `Cursor` to rows and columns

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In many cases, though, the complexity of managing groups and children will steer you down the path of extending `BaseExpandableListAdapter` and handling all of the view construction yourself. There are many methods that you will need to implement:

- `getGroupCount()`, to return the number of groups
- `getGroup()` and `getGroupId()`, to return an `Object` and unique `int` ID for a group given its position
- `getGroupView()`, to return the `View` that should be used to render the group, perhaps using the built-in `android.R.layout.simple_expandable_list_item_1` that is set up for such groups and handles rendering the expanded and collapsed states
- `getChildrenCount()`, to return the number of children for a given group
- `getChild()` and `getChildId()`, to return an `Object` and unique `int` ID for a child given its position (and its group's position)
- `getChildView()`, to return the `View` that should be used to render the child, given its position and its group's position
- `isChildSelectable()`, to indicate if the user can select a given child, given its position and its group's position
- `hasStableIds()`, to indicate if the ID values you returned from `getGroupId()` and `getChildId()` will remain constant for the life of this adapter

There are four major events that you will be able to respond to with respect to the user's interaction with an `ExpandableListView`:

- Clicks on a child (`setOnChildClickListener()`)
- Clicks on a group (`setOnGroupClickListener()`)
- When groups expand (`setOnGroupExpandListener()`) or collapse (`setOnGroupCollapseListener()`)

If you use `setOnGroupClickListener()` to be notified about clicks on a group, be sure to return `false` from your implementation of the `onGroupClick()` method required by the `OnGroupClickListener` interface. If you return `true`, you consume the click event, which prevents `ExpandableListView` from using that event to expand or collapse the group.

A Sample Usage

The sample project can be found in WidgetCatalog/ExpandableListView.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Layout:

```
<ExpandableListView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/elv"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

</ExpandableListView>
```

JSON data:

```
{
  "Group A": ["Child A1", "Child A2", "Child A3"],
  "Group B": ["Child B1", "Child B2"],
  "Group C": ["Child C1"],
  "Group D": [],
  "Group E": ["Child E1", "Child E2", "Child E3"]
}
```

Activity:

```
package com.commonsware.android.wc.elv;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.ExpandableListAdapter;
import android.widget.ExpandableListView;
import android.widget.ExpandableListView.OnChildClickListener;
import android.widget.ExpandableListView.OnGroupClickListener;
import android.widget.ExpandableListView.OnGroupCollapseListener;
import android.widget.ExpandableListView.OnGroupExpandListener;
import android.widget.Toast;
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import org.json.JSONObject;

public class MainActivity extends Activity implements
    OnChildClickListener, OnGroupClickListener, OnGroupExpandListener,
    OnGroupCollapseListener {
  private ExpandableListAdapter adapter=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    InputStream raw=getResources().openRawResource(R.raw.sample);
    BufferedReader in=new BufferedReader(new InputStreamReader(raw));
    String str;
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    StringBuffer buf=new StringBuffer();

    try {
      while ((str=in.readLine()) != null) {
        buf.append(str);
        buf.append('\n');
      }

      in.close();

      JSONObject model=new JSONObject(buf.toString());

      ExpandableListView elv=(ExpandableListView)findViewById(R.id.elv);

      adapter=new JSONExpandableListAdapter(getLayoutInflater(), model);
      elv.setAdapter(adapter);

      elv.setOnChildClickListener(this);
      elv.setOnGroupClickListener(this);
      elv.setOnGroupExpandListener(this);
      elv.setOnGroupCollapseListener(this);
    }
    catch (Exception e) {
      Log.e(getClass().getName(), "Exception reading JSON", e);
    }
  }

  @Override
  public boolean onChildClick(ExpandableListView parent, View v,
                              int groupPosition, int childPosition,
                              long id) {
    Toast.makeText(this,
                   adapter.getChild(groupPosition, childPosition)
                          .toString(), Toast.LENGTH_SHORT).show();

    return(false);
  }

  @Override
  public boolean onGroupClick(ExpandableListView parent, View v,
                              int groupPosition, long id) {
    Toast.makeText(this, adapter.getGroup(groupPosition).toString(),
                   Toast.LENGTH_SHORT).show();

    return(false);
  }

  @Override
  public void onGroupExpand(int groupPosition) {
    Toast.makeText(this,
                   "Expanding: "
                       + adapter.getGroup(groupPosition).toString(),
                   Toast.LENGTH_SHORT).show();
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void onGroupCollapse(int groupPosition) {
    Toast.makeText(this,
                   "Collapsing: "
                       + adapter.getGroup(groupPosition).toString(),
                   Toast.LENGTH_SHORT).show();
  }
}
```

This activity loads up a JSON file from a raw resource on the main application thread in `onCreate()`, which is not a good idea. It would be better to do that work in a background thread, perhaps an `AsyncTask` managed by a retained fragment. The implementation shown here is designed to keep the sample small, not to demonstrate the best way to load data from a raw resource.

Adapter:

```
package com.commonsware.android.wc.elv;

import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.BaseExpandableListAdapter;
import android.widget.TextView;
import java.util.Iterator;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

public class JSONExpandableListAdapter extends
    BaseExpandableListAdapter {
  LayoutInflater inflater=null;
  JSONObject model=null;

  JSONExpandableListAdapter(LayoutInflater inflater, JSONObject model) {
    this.inflater=inflater;
    this.model=model;
  }

  @Override
  public int getGroupCount() {
    return(model.length());
  }

  @Override
  public Object getGroup(int groupPosition) {
    @SuppressWarnings("rawtypes")
    Iterator i=model.keys();
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    while (groupPosition > 0) {
      i.next();
      groupPosition--;
    }

    return(i.next());
  }

  @Override
  public long getGroupId(int groupPosition) {
    return(groupPosition);
  }

  @Override
  public View getGroupView(int groupPosition, boolean isExpanded,
                           View convertView, ViewGroup parent) {
    if (convertView == null) {
      convertView=
          inflater.inflate(android.R.layout.simple_expandable_list_item_1,
                           parent, false);
    }

    TextView tv=
        ((TextView)convertView.findViewById(android.R.id.text1));
    tv.setText(getGroup(groupPosition).toString());

    return(convertView);
  }

  @Override
  public int getChildrenCount(int groupPosition) {
    try {
      JSONArray children=getChildren(groupPosition);

      return(children.length());
    }
    catch (JSONException e) {
      // JSONArray is really annoying
      Log.e(getClass().getSimpleName(), "Exception getting children", e);
    }

    return(0);
  }

  @Override
  public Object getChild(int groupPosition, int childPosition) {
    try {
      JSONArray children=getChildren(groupPosition);

      return(children.get(childPosition));
    }
    catch (JSONException e) {
      // JSONArray is really annoying
      Log.e(getClass().getSimpleName(),
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
            "Exception getting item from JSON array", e);
    }

    return(null);
  }

  @Override
  public long getChildId(int groupPosition, int childPosition) {
    return(groupPosition * 1024 + childPosition);
  }

  @Override
  public View getChildView(int groupPosition, int childPosition,
                           boolean isLastChild, View convertView,
                           ViewGroup parent) {
    if (convertView == null) {
      convertView=
          inflater.inflate(android.R.layout.simple_list_item_1, parent,
                           false);
    }

    TextView tv=(TextView)convertView;
    tv.setText(getChild(groupPosition, childPosition).toString());

    return(convertView);
  }

  @Override
  public boolean isChildSelectable(int groupPosition, int childPosition) {
    return(true);
  }

  @Override
  public boolean hasStableIds() {
    return(true);
  }

  private JSONArray getChildren(int groupPosition) throws JSONException {
    String key=getGroup(groupPosition).toString();

    return(model.getJSONArray(key));
  }
}
```

This adapter wraps a `JSONObject` and assumes that the JSON structure is an object, keyed by strings, whose values are arrays of strings. The object returned by `getGroup()` is the key for that group's position; the object returned by `getChild()` is the string at that child's array index for it's group's array. Since the data structure is treated as immutable, and since there are no other better IDs in the data structure itself, the group ID is simply the group's position, and the child's ID is simply a mash-up of the group and child positions.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Visual Representation

This is what an `ExpandableListView` looks like in a few different Android versions and configurations, based upon the sample app shown above.

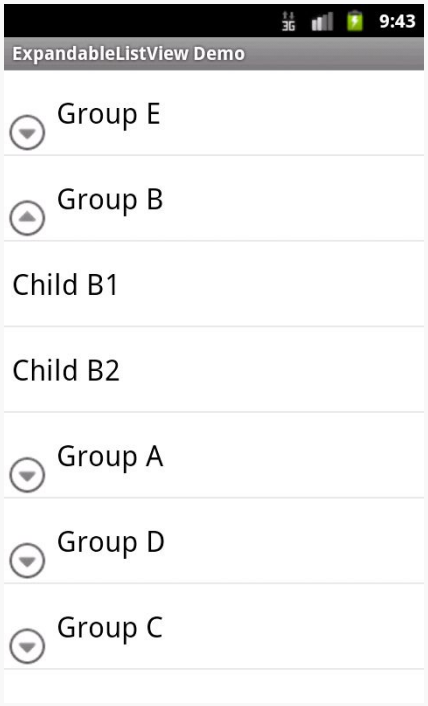


Figure 833: Android 2.3.3, Portrait

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

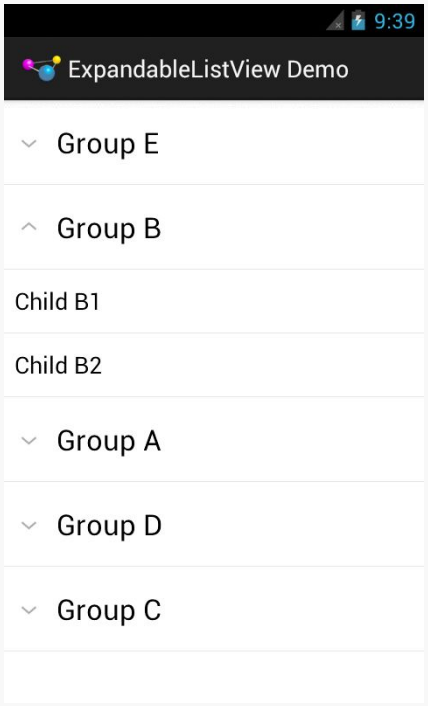


Figure 834: Android 4.0.3, Portrait

Note that while the data in the JSON file has the groups sorted alphabetically, because `JSONObject` effectively loads its data into a `HashMap`, the sorting gets lost in the data model, which is why the groups appear out of order.

Also note that the visual representation of the "collapsed" and "expanded" states is controlled by the `ExpandableListAdapter` and the view used for the groups. In this sample, we use `android.R.layout.simple_expandable_list_item_1` for the groups, which gives us the caret designation for expanded versus collapsed states in 4.0.3 and the lower-left arrowhead-in-circle icon for 2.3.3. You can create your own rows with your own indicators as you see fit.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Widget Catalog: SeekBar

`SeekBar` allows the user to choose a value along a continuous range by sliding a "thumb" along a horizontal line. In effect — and in practice, as it turns out – `SeekBar` is a user-modifiable `ProgressBar`.

Key Usage Tips

The value range of a `SeekBar` runs from 0 to a developer-set maximum value. As with `ProgressBar`, the default maximum is 100, but that can be changed via an `android:max` attribute or the `setMax()` method. The minimum value is always 0, so if you want a range starting elsewhere, just add your starting value to the actual value (obtained via `getProgress()`) to slide the range as desired.

You can find out about changes in the `SeekBar` value by attaching an `OnSeekBarChangeListener` implementation. The primary method on that interface is `onProgressChanged()`, where you are notified about changes in the progress value (second parameter) and whether that change was initiated directly by the user interacting with the widget (third parameter). The interface also has `onStartTrackingTouch()` and `onStopTrackingTouch()`, to indicate when the user is attempting to change the position of the thumb via the touchscreen, though these methods are less-commonly used.

A Sample Usage

The sample project can be found in `WidgetCatalog/SeekBar`.

Layout:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:gravity="center_vertical"
  tools:context=".MainActivity">

  <TextView
    android:id="@+id/value"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="0"
    android:ems="2"
    android:gravity="right|center_vertical"
    android:layout_marginRight="10dp"
    android:textAppearance="@android:style/TextAppearance.Large"/>

  <SeekBar
    android:id="@+id/seek_bar"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:layout_marginRight="10dp"
    android:max="50"/>

</LinearLayout>
```

Activity:

```
package com.commonsware.android.wc.seekbar;

import android.app.Activity;
import android.os.Bundle;
import android.widget.SeekBar;
import android.widget.SeekBar.OnSeekBarChangeListener;
import android.widget.TextView;

public class MainActivity extends Activity implements
    OnSeekBarChangeListener {
  TextView value=null;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    value=(TextView)findViewById(R.id.value);

    SeekBar seekBar=(SeekBar)findViewById(R.id.seek_bar);

    seekBar.setOnSeekBarChangeListener(this);
  }
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
  @Override
  public void onProgressChanged(SeekBar seekBar, int progress,
                                boolean fromUser) {
    value.setText(String.valueOf(progress));
  }

  @Override
  public void onStartTrackingTouch(SeekBar seekBar) {
    // no-op
  }

  @Override
  public void onStopTrackingTouch(SeekBar seekBar) {
    // no-op
  }
}
```

Visual Representation

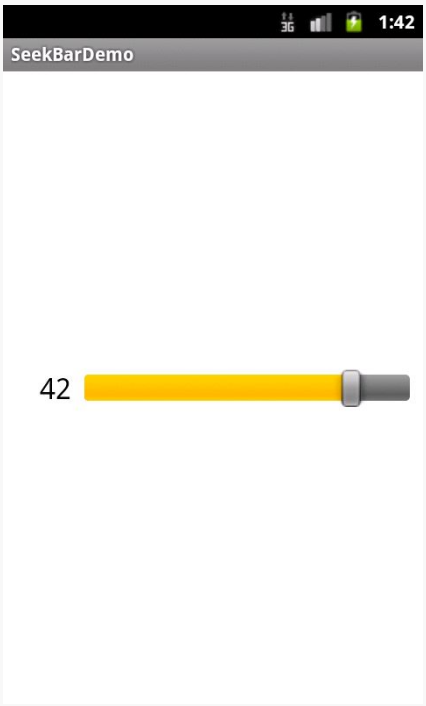


Figure 835: Android 2.3.3

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

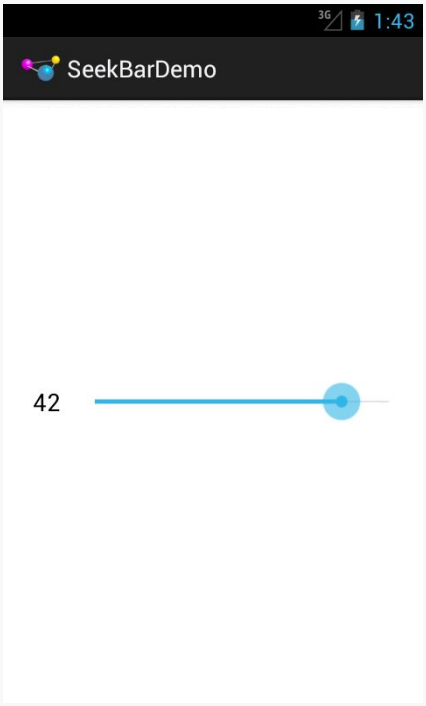


Figure 836: Android 4.1

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Widget Catalog: SlidingDrawer

Having some form of means of allowing the user to swipe to show more things is an important visual pattern. We saw this earlier in the book with the `ViewPager` container. And there are other modern techniques for doing this that you will see in apps like Google+.

`SlidingDrawer`, while implementing a variation on this pattern, is a bit out of date at present. Mostly, that's a question of its UI: tapping a drawer "handle" to open it is not what you tend to see nowadays. That being said, it works perfectly well, wrapping around a container to make it appear or disappear based on user input, complete with a sliding animation effect.

Note that `SlidingDrawer` was deprecated in API Level 17 (a.k.a., Android 4.2). This means that Google is steering you in other directions, including forking the AOSP code for `SlidingDrawer` and maintaining it yourself. The animator framework offers other ways of implementing sliding widgets that may be better suited for your UI, anyway.

Also note that `SlidingDrawer` is broken on Android 5.0, and so you *definitely* should be considering alternative widgets at this time.

Key Usage Tips

The `SlidingDrawer` itself is transparent, except for the button to trigger the slide and its accompanying horizontal bar. Hence, if you want the drawer contents to completely obscure what is outside of the drawer, you will need to use an appropriate background. Otherwise, the drawer contents and what lies outside the drawer will be alpha-blended based on their own translucency, as is seen in the screenshots later in this chapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The `SlidingDrawer` can be horizontal or vertical; it is vertical by default. However, it only slides one way (bottom-to-top for vertical, right-to-left for horizontal). There is no way to reverse the direction of the sliding effect.

You must supply `android:content` and `android:handle` attributes in `SlidingDrawer`, containing references to the widget that forms the content of the drawer and the drawer's handle, respectively. Typically, the drawer's handle is an `ImageView`. Note that you *must* supply a handle — you cannot skip either of these attributes.

A Sample Usage

The sample project can be found in [WidgetCatalog/SlidingDrawer]().

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:text="@string/drawer_closed"/>

  <SlidingDrawer
    android:id="@+id/drawer"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:content="@+id/content"
    android:handle="@+id/handle">

    <ImageView
      android:id="@id/handle"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/tray_handle_normal"/>

    <Button
      android:id="@id/content"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:text="@string/drawer_msg"/>
  </SlidingDrawer>

</RelativeLayout>
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Activity:

```
package com.commonsware.android.drawer;

import android.app.Activity;
import android.os.Bundle;

public class DrawerDemo extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Visual Representation

This is what a `SlidingDrawer` looks like in a few different Android versions and configurations, based upon the sample app shown above.

Figure 837: Android 2.3.3, with Drawer Closed

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

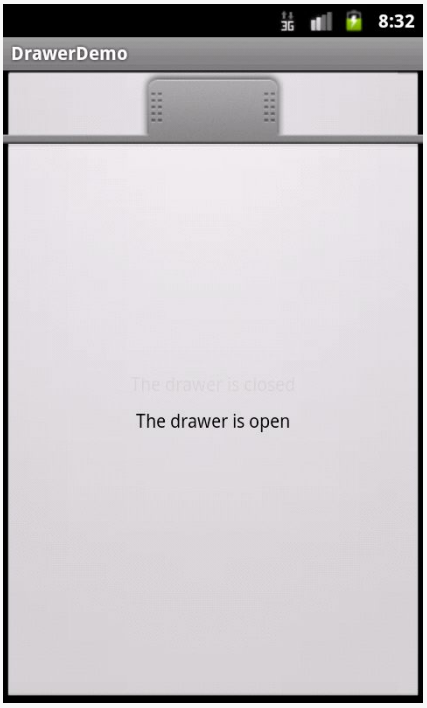


Figure 838: Android 2.3.3, with Drawer Open

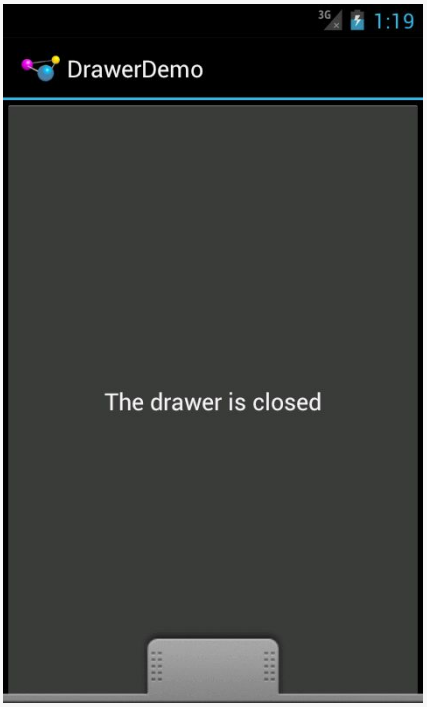


Figure 839: Android 4.0.3, with Drawer Closed

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Widget Catalog: StackView

`StackView` is an `AdapterView`. Whereas `ListView` uses a horizontal scrolling list as its UI metaphor, `StackView` uses a stack of cards as its metaphor. Just as `ListView` shows a handful of rows, `StackView` shows a handful of cards. These cards can be swiped away via a swipe towards the southwest corner of the screen. The top card is fully visible; the edges of a few other cards can be seen but are otherwise obscured by cards "higher in the stack".

While certainly usable in activities and fragments, `StackView` was introduced in support of app widgets. App widgets like bookmarks, Google Books covers, and the like use `StackView` to show an item and allow users to navigate to the rest of the items by flipping these virtual cards.

Key Usage Tips

Generally speaking, working with `StackView` is not significantly different than is working with any other `AdapterView`. You create an `Adapter` defining the contents (in this case, defining the cards), you attach the `Adapter` to the `StackView`, and put the `StackView` somewhere on the screen.

As the cards overlap, however, transparency becomes an issue. If the top card is not completely opaque, you will see the card beneath it "peeking through" as its contents are blended in via the alpha channel. In some cases, this is a perfectly desirable outcome. However, if that is not what you want, make sure that the backgrounds of your overall container for the card's contents (e.g., a `RelativeLayout`) has an opaque background, such as a color with `FF` for the alpha value.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Also, since the objective is to have the children be visually stacked, the children cannot be the size of the `StackView` itself (e.g., the children cannot use `match_parent` for a dimension). `StackView` seems to work best with children that have explicit sizes (e.g., values in `dp`).

A Sample Usage

The sample project can be found in WidgetCatalog/StackView.

Activity Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<StackView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/details"
  android:layout_width="match_parent"
  android:layout_height="match_parent"/>
```

Item Layout:

```
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="200dp"
  android:layout_height="200dp"
  android:background="#FFFF0000"
  android:gravity="center"
  android:textAppearance="?android:attr/textAppearanceLarge"/>
```

Activity:

```
package com.commonsware.android.wc.stack;

import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.StackView;

public class MainActivity extends Activity {
  static String[] items= { "lorem", "ipsum", "dolor", "sit", "amet",
      "consectetuer", "adipiscing", "elit", "morbi", "vel", "ligula",
      "vitae", "arcu", "aliquet", "mollis", "etiam", "vel", "erat",
      "placerat", "ante", "porttitor", "sodales", "pellentesque",
      "augue", "purus" };
  StackView stack;

  @Override
  public void onCreate(Bundle icicle) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onCreate(icicle);
    setContentView(R.layout.main);

    stack=(StackView)findViewById(R.id.details);
    stack.setAdapter(new ItemAdapter(this, R.layout.item, items));
  }

  private static class ItemAdapter extends ArrayAdapter<String> {
    public ItemAdapter(Context context, int textViewResourceId,
                       String[] objects) {
      super(context, textViewResourceId, objects);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      View result=super.getView(position, convertView, parent);

      result.setBackgroundColor(0xFF330000 + (position * 0x0A0A));

      return(result);
    }
  }
}
```

Visual Representation

This is what a `StackView` looks like in Android 4.0.3, based upon the sample app shown above:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

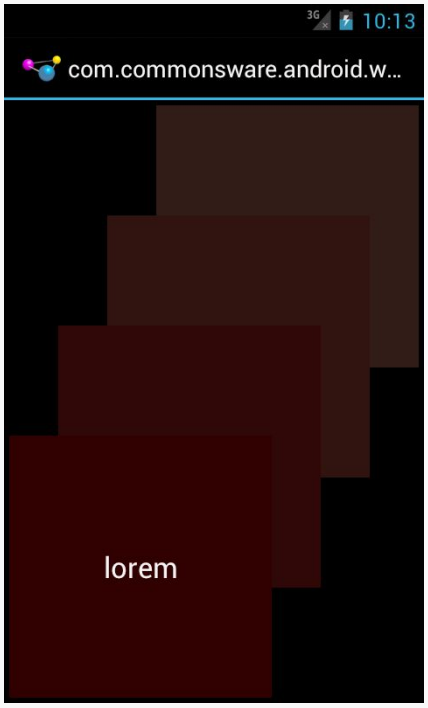


Figure 840: Android 4.0.3, As Initially Seen

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Widget Catalog: TabHost and TabWidget

Before we had the action bar and `ViewPager`, we had `TabHost` and `TabWidget` as our means of displaying tabs. Nowadays, in most cases, using tabs in the action bar would be preferable, or perhaps using "swipey tabs" with `ViewPager`. However, there may be cases where the classic tabs are a better solution, or you may have inherited legacy code that still uses `TabHost`.

Deprecation Notes

Just as `ListActivity` helps one use a `ListView`, `TabActivity` helps one use a `TabHost`. However, `TabActivity` is marked as deprecated. That is largely because its parent class, `ActivityGroup`, is deprecated. While you can still use `TabActivity`, it is no longer recommended. It also is not necessary, as there are ways to use `TabHost` and `TabWidget` without using `TabActivity`, as will be demonstrated later in this chapter.

Key Usage Tips

There are a few widgets and containers you need to use in order to set up a tabbed portion of a view:

- `TabHost` is the overarching container for the tab buttons and tab contents
- `TabWidget` implements the row of tab buttons, which contain text labels and optionally contain icons
- `FrameLayout` is the container for the tab contents; each tab content is a child of the `FrameLayout`

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

You load contents into that `FrameLayout` in one of two ways:

1. You can define the contents simply as child widgets (or containers) of the `FrameLayout` in a layout XML file you are using for the whole tab setup
2. You can define the contents at runtime

Curiously, you do not define what goes in the tabs themselves, or how they tie to the content, in the layout XML file. Instead, you must do that in Java, by creating a series of `TabSpec` objects (obtained via `newTabSpec()` on `TabHost`), configuring them, then adding them in sequence to the `TabHost` via `addTab()`.

The two key methods on `TabSpec` are:

- `setContent()`, where you indicate what goes in the tab content for this tab, typically the `android:id` of the view you want shown when this tab is selected
- `setIndicator()`, where you provide the caption for the tab button and, in some flavors of this method, supply a `Drawable` to represent the icon for the tab

Note that tab "indicators" can actually be views in their own right, if you need more control than a simple label and optional icon.

Also note that you must call `setup()` on the `TabHost` before configuring any of these `TabSpec` objects. The call to `setup()` is not needed if you are using the `TabActivity` base class for your activity.

A Sample Usage

The sample project can be found in WidgetCatalog/Tab.

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/tabhost"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TabWidget android:id="@android:id/tabs"
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
    />
    <FrameLayout android:id="@android:id/tabcontent"
      android:layout_width="match_parent"
      android:layout_height="match_parent">
      <AnalogClock android:id="@+id/tab1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
      />
      <Button android:id="@+id/tab2"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="A semi-random button"
      />
    </FrameLayout>
  </LinearLayout>
</TabHost>
```

Activity:

```
package com.commonsware.android.tabhost;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TabHost;

public class TabDemo extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    TabHost tabs=(TabHost)findViewById(R.id.tabhost);

    tabs.setup();

    TabHost.TabSpec spec=tabs.newTabSpec("tag1");

    spec.setContent(R.id.tab1);
    spec.setIndicator("Clock");
    tabs.addTab(spec);

    spec=tabs.newTabSpec("tag2");
    spec.setContent(R.id.tab2);
    spec.setIndicator("Button");
    tabs.addTab(spec);
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that ordinarily you would use icons with your tabs, and so the second parameter to `setIndicator()` would be a reference to a drawable resource. This particular sample skips the icons.

Visual Representation

This is what a `TabHost` and `TabWidget` look like in a few different Android versions and configurations, based upon the sample app shown above.

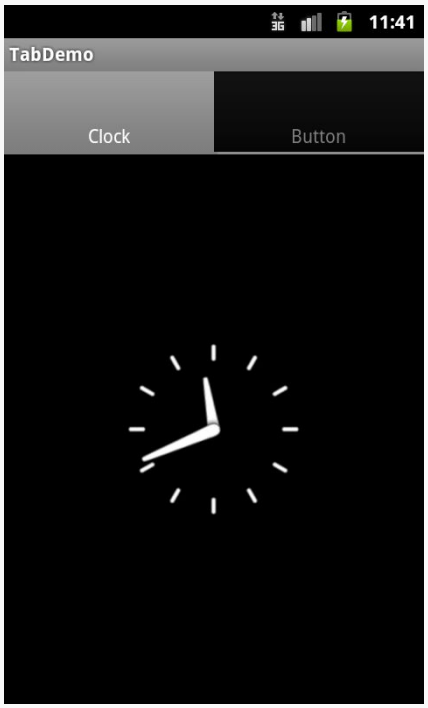


Figure 841: Android 2.3.3

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

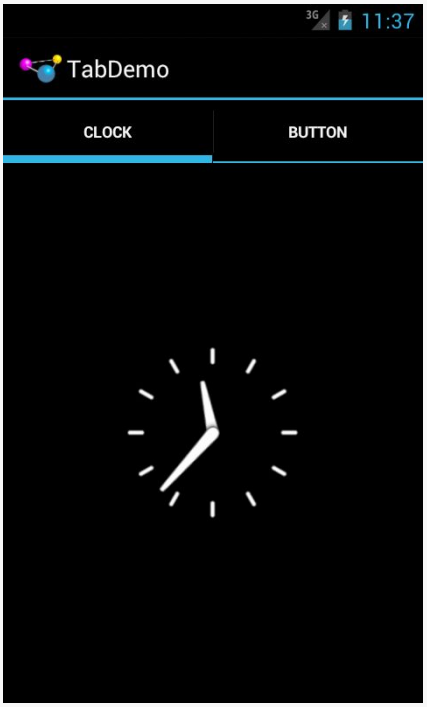


Figure 842: Android 4.0.3

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Widget Catalog: TimePicker

Just as `DatePicker` allows the user to pick a date, `TimePicker` allows the user to pick a time. This widget is a bit simpler to use, insofar as you do not have the option of the integrated `CalendarView` as you do with `DatePicker`. In other respects, `TimePicker` follows the patterns established by `DatePicker`.

Note that `TimePicker` only supports hours and minutes, not seconds or finer granularity.

Key Usage Tips

With `DatePicker`, the act of supplying an `OnDateSetListener` also required you to supply the year/month/day to use as a starting point. `TimePicker` is more intelligently designed: setting the `OnTimeSetListener` is independent from adjusting the hour or minute.

As with `DatePicker`, `TimePicker` works well with `Calendar` and `GregorianCalendar`, in terms of setting and getting the hour/minute/second from the `TimePicker` and converting it into something you can use in your code.

There is a bug in Android 4.0/4.0.3 in which your `OnTimeSetListener` is not invoked when the user changes between AM and PM when viewing the `TimePicker` in 12-hour display mode.

A Sample Usage

The sample project can be found in `WidgetCatalog/TimePicker`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:gravity="center_vertical">

  <TimePicker
    android:id="@+id/picker"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>

</LinearLayout>
```

Activity:

```
package com.commonsware.android.wc.timepick;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TimePicker;
import android.widget.TimePicker.OnTimeChangedListener;
import android.widget.Toast;
import java.util.Calendar;

public class TimePickerDemoActivity extends Activity implements
    OnTimeChangedListener {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    TimePicker picker=(TimePicker)findViewById(R.id.picker);

    picker.setOnTimeChangedListener(this);
  }

  @Override
  public void onTimeChanged(TimePicker view, int hourOfDay, int minute) {
    Calendar then=Calendar.getInstance();

    then.set(Calendar.HOUR_OF_DAY, hourOfDay);
    then.set(Calendar.MINUTE, minute);
    then.set(Calendar.SECOND, 0);

    Toast.makeText(this, then.getTime().toString(), Toast.LENGTH_SHORT)
         .show();
  }
}
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Visual Representation

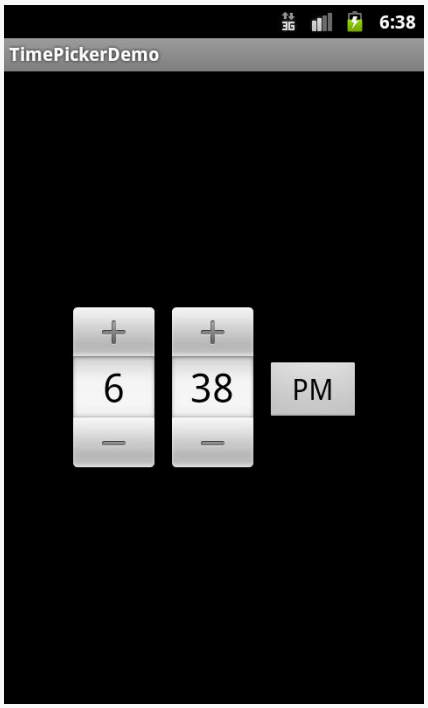


Figure 843: Android 2.3.3

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

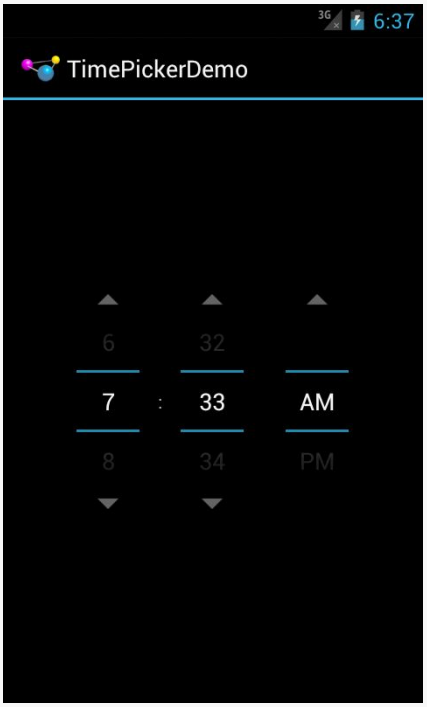


Figure 844: Android 4.0.3

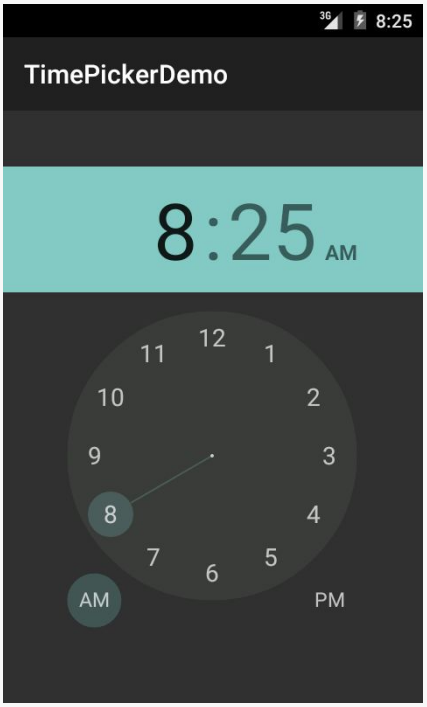


Figure 845: Android 5.0

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Widget Catalog: ViewFlipper

A `ViewFlipper` behaves a bit like a `FrameLayout` that is set up such that only one child can be visible at a time. You can control which of those children is visible, either by index or via `showNext()`/`showPrevious()` methods to rotate between them.

You can also set up animated effects to control how a child leaves and the next one enters, such as applying a sliding effect.

And, you can set up `ViewFlipper` to automatically flip between children on a specified period, without further developer involvement. This, coupled with the animation, can be used for news tickers, ad banner rotations, or the like where light animations (e.g., fade out and fade in) can be used positively.

Key Usage Tips

`ViewFlipper` can have as many children as needed (within memory constraints), though you will want at least two for it to be meaningful.

By default, the transition between children is an immediate "smash cut" — the old one vanishes and the new one appears instantaneously. You can call `setInAnimation()` and/or `setOutAnimation()` to supply an `Animation` object or resource to use for the transitions instead.

By default, the `ViewFlipper` will show its first child and stay there. You can manually flip children via `showNext()`, `showPrevious()`, and `setDisplayedChild()`, the latter of which taking a position index of which child to display. You can also have automatic flipping, by one of two means:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. In your layout, `android:flipInterval` will set up the amount of time to display each child before moving to the next, and `android:autoStart` will indicate if the automated flipping should begin immediately or not
2. In Java, `setFlipInterval()` serves the same role as `android:flipInterval`, and you can control when flipping is enabled via `startFlipping()` and `stopFlipping()`

A Sample Usage

The sample project can be found in WidgetCatalog/ViewFlipper.

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <ViewFlipper android:id="@+id/details"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  </ViewFlipper>
</LinearLayout>
```

Activity:

```
package com.commonsware.android.flipper2;

import android.app.Activity;
import android.os.Bundle;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.ViewFlipper;

public class FlipperDemo2 extends Activity {
  static String[] items={"lorem", "ipsum", "dolor", "sit", "amet",
                         "consectetuer", "adipiscing", "elit",
                         "morbi", "vel", "ligula", "vitae",
                         "arcu", "aliquet", "mollis", "etiam",
                         "vel", "erat", "placerat", "ante",
                         "porttitor", "sodales", "pellentesque",
                         "augue", "purus"};
  ViewFlipper flipper;

  @Override
  public void onCreate(Bundle icicle) {
```


Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

```
    super.onCreate(icicle);
    setContentView(R.layout.main);

    flipper=(ViewFlipper)findViewById(R.id.details);

    for (String item : items) {
      Button btn=new Button(this);

      btn.setText(item);

      flipper.addView(btn,
                      new ViewGroup.LayoutParams(
                              ViewGroup.LayoutParams.FILL_PARENT,
                              ViewGroup.LayoutParams.FILL_PARENT));
    }

    flipper.setFlipInterval(2000);
    flipper.startFlipping();
  }
}
```

Visual Representation

There is no visual representation of a `ViewFlipper` itself, as it renders no pixels on its own. Rather, it simply shows the current child.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Device Catalog

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Device Catalog: Kindle Fire

The most aggressive firm in creating a Google-free Android ecosystem is Amazon. The most visible aspect of that work is Amazon's Kindle Fire series of devices. From 2011 through 2013, each year has brought forth a new generation of Kindle Fire models, each with newer versions of Android and more powerful hardware. However, none support the Play Store or other Google proprietary apps and technologies. As a result, Amazon has also been building their own replacements, which developers can elect to use if they need the capabilities but are targeting the Fire.

This chapter will outline what you should expect as you start working on apps for the Kindle Fire series of devices.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

Introducing the Kindle Fire series

Once upon a time, there was the Kindle Fire. It was a 7" tablet, made by Amazon.

Nowadays, there is a series of devices in the Kindle Fire family:

- The original first-generation Kindle Fire
- The second-generation Kindle Fire, with an updated OS and slightly faster CPU
- The Kindle Fire HD 7", with a 720p display

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

- The Kindle Fire HD 8.9" with a 1080p display (available as a WiFi-only device or with mobile data capability)
- The Kindle Fire HDX 7", with a 1920x1200 display and optional mobile data
- The Kindle Fire HDX 8.9", with a 2560x1600 display and optional mobile data

This chapter will attempt to point out which of these devices certain statements pertain to. The phrase "the original Kindle Fire" refers to the first-generation Kindle Fire.

What Features and Configurations Does It Use?

Any time you are looking at a device that is known to be a significant departure from conventional Android devices, you need to consider what capabilities the device has and how that relates to your code and graphic assets. Android's flexibility means that, in many cases, you can work within the limits of the SDK to craft something that will look well on unusual devices. However, you will need to understand what is and is not possible for the device in question, in this case the Kindle Fire.

However, there are now several devices in the Kindle Fire family, which makes this more complicated.

OS Version

Amazon has branded their variation of Android "FireOS".

The first-generation Kindle Fire runs an Amazon-customized version of Android 2.3.3. The HD models run an Amazon-customized version of Android 4.0.3, while the HDX models run a FireOS based on Android 4.2.2.

Note that these devices will not show up in Google's "Device Dashboard" pie chart, as Google can only count those devices that use the Play Store, which the Kindle Fire series lacks.

Screen Size and Density

The original Kindle Fire (first and second generation) uses `-large`, `-mdpi` resources. On the surface, this would not be terribly surprising, as the 7" display works out to around 169dpi, and 7" displays are definitely in the `-large` resource bucket.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, bear in mind that Android 2.3 did not fully support tablets. The only Google-endorsed tablet that shipped with Android 2.x was the original Samsung Galaxy Tab, and that technically was a really large phone that, er, could not place phone calls.

As such, Android 2.3 did not consider a 1024x600 display to be `-large`. It considered such a display to be `-xlarge`. This was corrected in Android 3.1, in preparation for a new line of ~7" Honeycomb tablets.

In general, this should not pose an issue when testing your app on hardware. In practice, it will pose a problem for your emulator, as will be explained later in this chapter.

The Kindle Fire HD 7" and 8.9" are `-large`, though they are `-hdpi` in terms of screen density.

The Kindle Fire HDX 7" is `-large`, while the Kindle Fire HDX 8.9" is `-xlarge`. Both are `xhdpi` in terms of screen density.

Hardware Features

All of the Kindle Fire devices support:

1. An accelerometer, both for direct use and for detecting screen orientation changes
2. Multitouch, but only for two fingers (e.g., pinch-to-zoom)
3. WiFi
4. The USB accessory interface
5. A light/proximity sensor

The Kindle Fire HD devices add:

1. Front-facing camera
2. Network-based location (the Kindle Fire HD 8.9", with the mobile data option, also has GPS)
3. Bluetooth
4. Microphone

The Kindle Fire HDX devices add:

1. GPS

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. Compass

The Kindle Fire HDX 8.9" offers a rear-facing camera in addition to the front-facing camera.

None of the Kindle Fire series support telephony (voice or SMS).

If your application truly needs any of those missing capabilities, you are out of luck.

If your application could use some of those capabilities but can get by without them, be sure to add the appropriate `<uses-feature>` elements to your manifest with `android:required="false"` (e.g., `<uses-feature android:name="android.hardware.camera" android:required="false" />`). Otherwise, your app will not be available for the original Kindle Fire if Android thinks that you really do need the capability (e.g., you have requested the `CAMERA` permission).

What Is Really Different?

All of the devices profiled in this part of the book are clearly different than what you are used to from an Android development standpoint. Some things, like availability of Bluetooth, will fit within the Android SDK's framework for optional capabilities. Other things will represent where a device manufacturer has meandered farther from the Android device norm, in ways that may not be completely obvious to you, let alone your code.

The Menu Bar

As was noted previously in this chapter, the original version of the Kindle Fire runs Android 2.3, a version of Android not designed for tablets. Moreover, Android 2.3 was designed for devices that had dedicated off-screen options for HOME, BACK, and MENU buttons. However, Amazon apparently wanted to avoid such buttons, yet they lacked source code access to Honeycomb, where support for the system bar was added.

So, they faked it.

The Kindle Fire supports what Amazon refers to as the "menu bar". This is akin to the system bar found on tablets running Android 3.0+, insofar as:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

1. It appears at the bottom of the screen
2. It contains the HOME, BACK, and MENU buttons, along with a search button

However, unlike the system bar:

1. The menu bar disappears when not in use, in some cases
2. There is still a status bar at the top containing signal strength, battery level, time, etc.

Here, for example, is an application running on the original Kindle Fire:

Figure 846: The original Kindle Fire, running a sample application, showing the menu bar

In this case, this is a normal activity, and the menu bar is always visible.

However, here is the same activity with `android:theme="@android:style/Theme.NoTitleBar.Fullscreen"` in the manifest:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 847: The original Kindle Fire, running a sample application, with the menu bar collapsed

Hence, if you set your activity to be full-screen, the status bar at the top goes away, and the menu bar shrinks to a smaller bar. Tapping on that bar brings back the menu bar, but this time overlaying the bottom portion of your activity.

Nothing Googly

The Kindle Fire lacks Google Maps, both the app and the library used for things like `MapView`.

The Kindle Fire lacks the Play Store and anything that depends upon it, such as GCM.

The Kindle Fire lacks Gmail.

The Kindle Fire lacks anything from Google that is not part of the Android Open Source Project.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

If your application depends on one or more of these, your app will not work well on a Kindle Fire without adjustments, though some have alternatives from Amazon that can be used.

Sideloading Limitations

If you enable the standard Android setting, you can install apps on the Kindle Fire from alternative sources, such as sideloading via USB. This is how the development tools deploy apps to a device when you are working on your app, and anyone can use this technique so long as they have the Android SDK (or at least enough to provide `adb` access).

However, there is one notable limitation of sideloading: icon quality.

When you submit your app for distribution through the Amazon AppStore, you will upload what they refer to as the "thumbnail" image. This is a 512x512 pixel rendition of your icon and is independent from any icons you may have put as resources in the APK file itself. When your app is installed from the Amazon AppStore, your thumbnail is downloaded as well and is used for the home screen carousel, among other things:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

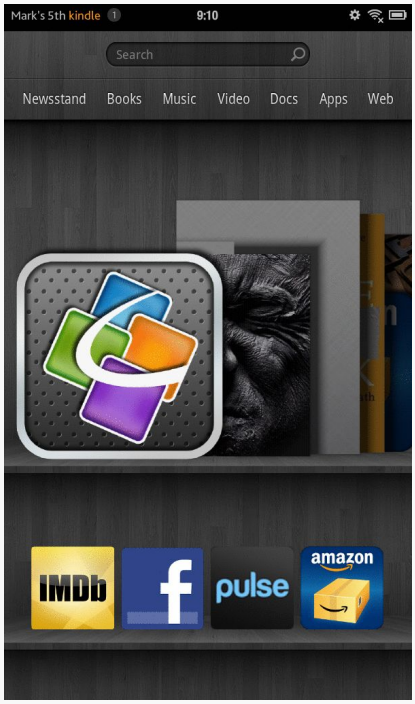


Figure 848: The original Kindle Fire home screen, with a high-resolution version of the QuickOffice icon

However, when you sideload an app, or install it off the Web, there is no "thumbnail". The Kindle Fire will use your in-APK icon, no different than any other home screen. However, when it blows up your, say, 72x72 pixel icon to the large shelf in the carousel, it does not look very pretty:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

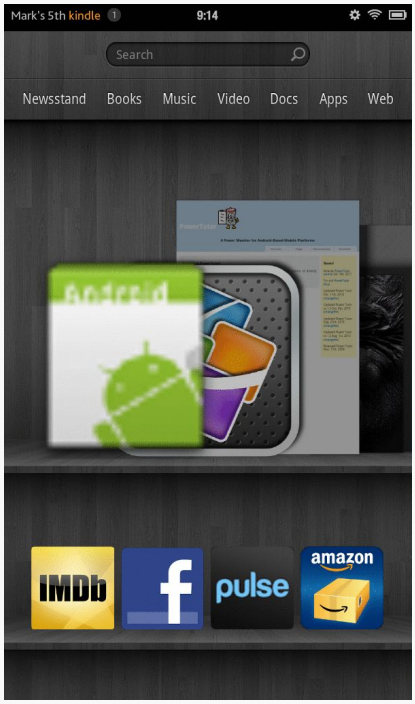


Figure 849: The original Kindle Fire home screen, with a not-so-high-resolution version of the stock Android launcher icon

Things are somewhat better on the HD series:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

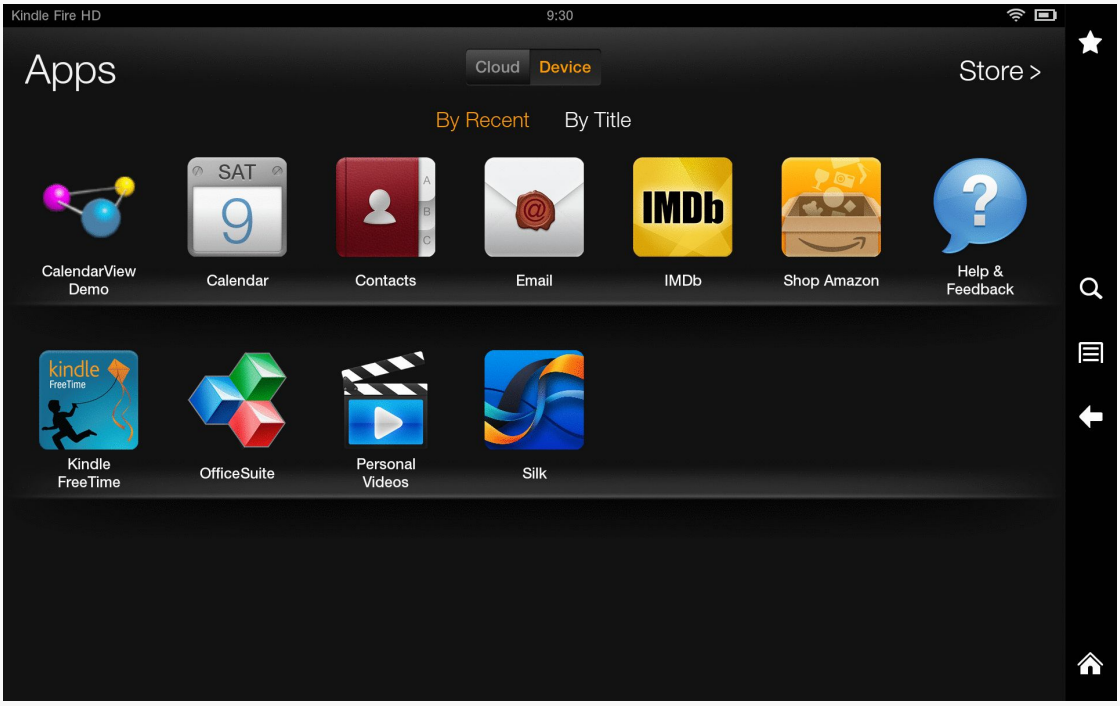


Figure 850: Kindle Fire HD, with CommonsWare App Icon

Getting Your Development Environment Established

Developing for the Kindle Fire series is best accomplished using an actual Kindle Fire device. For example, there is no good way to simulate the behavior of the Kindle Fire menu bar using the standard Android emulator. That being said, having an emulator that at least resembles the Kindle Fire will be useful for debugging purposes, since you can do more with an emulator (e.g., run Hierarchy View) than you can with production devices.

Emulator Configuration

Amazon does not distribute an emulator image for the Fire, meaning that developers have to fake it as best they could using a stock emulator. This is fairly limiting, as the Fire does not look much like a standard Android emulator.

Briefly, Amazon was distributing an SDK add-on with emulator images, but that SDK add-on no longer seems to offer such images.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Note that the emulators in portrait mode get a bit tall, in terms of pixels, so be sure to use the scaling option in the AVD Manager to scale down the emulator so that it will fit your development machine's monitor.

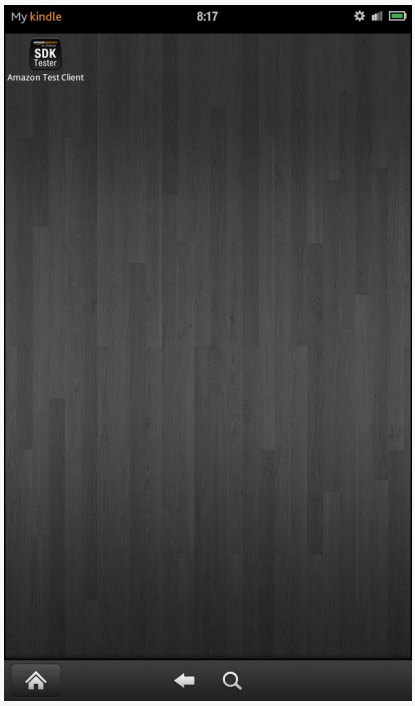


Figure 851: Kindle Fire Emulator

The official original Kindle Fire emulator image also overcomes a limitation in the standard Android emulator image.

As mentioned earlier in this chapter, Gingerbread did not support tablets. More importantly, it had a snippet of code that assumes that devices running with the Kindle Fire's resolution must be `-xlarge`. In reality, the Kindle Fire (and other 7" tablets) should use a `-large` configuration. However, the standard Android emulator will use `-xlarge`. However, the official Kindle Fire emulator will correctly report the emulator as `-large`, matching the device.

Developing on Hardware

The Kindle Fire is ready for use with your development tools, once you teach your development machine how to have `adb` connect to the fire.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Linux and OS X users simply need to run `android update adb`, after having installed the Kindle Fire SDK components, to have the ADB USB entries added to the `adb_usb.ini` file.

On Windows, you will need to do that too, after doing some other things to unpack a local copy of the device drivers. Details for this process can be found in the Kindle Fire developer documentation.

Note that the original Kindle Fire automatically switches into USB Mass Storage mode when you plug it into a PC using the USB cable. This means that apps on the Kindle Fire do not have access to external storage. You will need to unmount the Kindle from your development machine's OS and click the Disconnect button on the Kindle Fire's "You can now transfer files from your computer to Kindle" screen to be both connected via USB *and* allow apps access to external storage.

How Does Distribution Work?

Unlike the vast majority of Android devices, the Kindle Fire series lacks the Play Store. It is quite likely the most popular device ever shipped that does not include the Play Store, though it is far from the first. Hence, if you want your app to be available to Kindle Fire users, you will need to explore other ways of promoting and delivering the app.

Amazon AppStore

The primary way to reach Kindle Fire users is through the Amazon AppStore. This is Amazon's equivalent to the Play Store. And, unlike the Play Store, which is only available pre-installed on devices, any Android device can download an app client for the Amazon AppStore. That, coupled with Amazon's promotions like the "free app of the day", means that your app on the Amazon AppStore has reach beyond just the Kindle Fire series and future Amazon Android devices.

At a high level, publishing on the Amazon AppStore is not significantly different than publishing on the Play Store: you supply the APK and descriptive material to Amazon, and it gets listed. However, the devil, as they say, is in the details:

1. Your app will be reviewed by Amazon before publishing, and it may be rejected for the same sorts of reasons why apps are rejected from the iOS App Store, for anything from content concerns to poor programming practices

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

2. If you are trying to sell a paid app, Amazon holds final pricing decisions, and your prices on the Amazon AppStore cannot be higher than on other venues
3. Your app will be wrapped in Amazon-supplied code and re-signed by Amazon, so that if a non-Kindle Fire user uninstalls the Amazon AppStore client, your app will no longer run
4. And so on

This is not to say that distributing through the Amazon AppStore is intrinsically a bad idea. Because of some of these hurdles, plus the AppStore's much smaller user base, many developers are skipping it. This results in less competition and greater visibility for your app. However, you need to review all the Amazon AppStore developer rules and make decisions for yourself as to whether it makes sense for you and what you are trying to accomplish with the app.

In April 2013, Amazon also launched Coins, which is their replacement for Google's in-app purchasing model for the Play Store.

Alternatives

Because Amazon did not license the Play Store or other commercial components from Google, you cannot reach Kindle Fire users through the Market (except for those who install pirated versions of the Play Store client on their devices).

However, all other distribution vectors should work as they would on any other device. In addition to sideloading via USB, users can install apps off of the Web by visiting a URL in the device's browser (by default, Amazon Silk) and tapping on the link to the APK. This will trigger a download of the app — users can then tap on the `Notification` for the download to trigger an install. Similarly, one would imagine that other apps whose job is to download and install apps (e.g., enterprise app "markets") should work normally as well.

Note, though, that all off-AppStore installs will have rough icons, so you will want to supply your icons in all densities, in hopes that the Kindle Fire will choose a higher-quality rendition of the icon.

Amazon Equivalents of Google Services

Since Amazon does not license the Google proprietary apps, the Kindle Fire series lacks common things like Google Maps.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

However, Amazon has their own equivalents for some of these:

- The Amazon Maps API allows you to embed maps on Kindle Fire devices like you would embed Google Maps on more traditional Android devices
- The Amazon Device Messaging API is roughly analogous to GCM for pushing messages to a Kindle Fire device
- The Amazon In-App Purchasing API allows your apps to tie into Amazon Payments and the like for collecting fees from users inside your app

Getting Help with the Kindle Fire

Amazon maintains a set of documentation related to Kindle Fire and Kindle Fire HDX development, along with a set of forums for asking Amazon-specific development questions regarding the Kindle Fire or their various SDKs.

Amazon is also reported to monitor the `kindle-fire` tag on Stack Overflow.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Device Catalog: BlackBerry

BlackBerry — formerly Research In Motion (BlackBerry) — is known around the world for their phones and messaging services. In 2011, they leapt into the tablet arena with the BlackBerry Playbook, and the 2.0 version of the Playbook OS supported running carefully repackaged Android applications.

While the Playbook itself had modest success, the ability to distribute Android applications to BlackBerry devices continues with their BlackBerry 10 platform. Many developers have enjoyed success distributing their app through BlackBerry World (the distribution channel for apps to BlackBerry products).

This chapter will describe a bit about what is involved in getting your Android app to BlackBerry devices.

I Thought BlackBerry Had Their Own OS?

They do.

However, current versions of that OS — this chapter was written when version 10.2 was the latest shipping version — contain an Android runtime environment. BlackBerry OS can run Android apps alongside apps written natively for BlackBerry OS or running on other runtimes (e.g., Adobe AIR). This gives developers a wide range of ways to get their app onto modern BlackBerry devices. However, it does mean that our apps may have somewhat less direct access to hardware, as there is another layer between us and that hardware.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What Else Is Different?

At its core, BlackBerry is a device manufacturer, no different than any other manufacturer that you may have dealt with previously. The biggest difference is BlackBerry's ability to run Android applications that you prepare for their devices.

That being said, the world of BlackBerry is a *bit* different than what you may be used to.

Hardware

BlackBerry makes phones with a variety of capabilities, much as do other manufacturers. You should be writing your apps to support a range of device characteristics, such as screen size and density. That will help you with BlackBerry support, just as it helps you with support for other manufacturers' devices.

Note, though, that BlackBerry has some history which will affect their device designs, and your apps by extension. Notably, BlackBerry has been renowned for their hardware keyboards. While not all BlackBerry devices today have such keyboards, it is likely that BlackBerry will ship keyboard-equipped devices for some time to come.

This has three impacts upon you as a developer:

1. Do not assume that the user is using a soft keyboard. Usually, this is not a problem from a programming standpoint, though you may wish to take it into account in documentation.
2. Do not assume that the user always uses the touchscreen to navigate. Some BlackBerry users may use the hardware keyboard for navigation. This is particularly true for users who gravitate towards hardware keyboards for accessibility reasons. Your app should support proper focus to allow it to be navigated without accessing the touchscreen, to the greatest extent possible.
3. BlackBerry keyboards often have not been "slider" keyboards (i.e., ones that might slide under the display when not in use). Rather, they are always available, below the screen. This will often result in somewhat smaller screen sizes, and odd aspect ratios, compared to what you are used to. There simply is not enough room for a large touchscreen *and* an always-available physical keyboard without having an excessively large device. The BlackBerry Q10, for example, has a 720x720 resolution screen, and most Android developers do not encounter square screen resolutions. You will need to take this into

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

account, but only in cases where such an aspect ratio might cause you problems (e.g., full-screen image backgrounds).

BlackBerry OS 10.2

Beyond the hardware, the BlackBerry OS — and the Android runtime environment that executes our Android apps — puts some limits on what we can do in our apps. Of note:

- BlackBerry OS 10.2's Android environment uses Android 4.2 (API Level 17), so apps that have an `android:minSdkVersion` higher than that will not be eligible to run on 10.2 devices.
- Some classes will be non-functional, particularly those related to telephony capabilities. You cannot use `SmsManager`, for example.
- Some types of apps will not work well, because the Android runtime runs alongside other non-Android apps on the hardware. Replacement home screen implementations, for example, are unlikely to work. Similarly, background apps that display a UI (e.g., Facebook "chatheads"-style popups) using `SYSTEM_ALERT_WINDOW` are unlikely to work.
- App widgets are not supported. So long as the app widget is a non-essential feature of your app, this should not be a problem — after all, the user does not *have* to use your app widget on any Android device. However, if the *sole purpose* of your app is to provide an app widget, such an app will not be useful on BlackBerry.
- Accounts via `AccountManager` are apparently not supported.
- Not all connectivity is surfaced in the Android runtime from the underlying hardware. Bluetooth Low Energy (BLE), WiFiDirect, NFC, and direct USB access are all unavailable.

Note that at the time of this writing (mid–2014), a BlackBerry OS 10.3 version is being beta-tested. This is slated to lift some of these restrictions and limitations, including:

- The Android runtime will be based on Android 4.3, giving you access to API Level 18 classes and methods
- `AccountManager`, Bluetooth LE, and NFC support should be added

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Navigation

Like most Android tablets, BlackBerry devices offer little in the way of physical or off-screen navigation buttons. For example, there is no BACK button. However, a navigation bar will contain a BACK soft button for users. If your app takes over the full screen, this bar will not be there all the time, but a swipe down from the top of the screen should expose it.

Similarly, your menu will not be accessed via a MENU key, but rather via a downward swipe to expose the menu. This also means that any special MENU-button logic of yours may not work, if you are using the MENU button for things other than displaying the action bar overflow or other form of options menu.

Nothing Googly

As with other devices cited in this book, the BlackBerry series of devices lack support for Google's proprietary apps. For app developers, this means that you lack access to Google Play Services and the various APIs exposed by it, such as Maps V2 and Google Cloud Messaging.

BlackBerry does offer its replacement for GCM, in the form of the BlackBerry Push Service. However, for maps, they steer you towards using `geo:` `Intent` structures and `startActivity()`. BlackBerry OS 10.2.1 also supports apps using the old Maps V1 APIs, for which BlackBerry has created their own implementation, backed by OpenStreetMap.

If you are dependent upon other APIs offered by Google Play Services (e.g., `LocationClient), you will need to reconsider your use of those APIs if you wish to ship on devices that are outside the Google ecosystem.

Package Name Length

The BlackBerry Android runtime only supports package names of 29 characters or less. The build tools will fail if your package name is longer than 29 characters.

Note that this should only pertain to the "application ID" role of a package name, not the "hey, where does `R.java` get generated?" role of the package name. Hence, it should be possible to replace the application ID of your app via a Gradle for Android product flavor, to give yourself a shorter identifier while not breaking your source code references to resources.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

What Are We Making?

This might seem like an odd question. After all, the point of this chapter is to make an Android application that can run on BlackBerry devices.

Up until early 2014, that meant you had one option: write a BAR.

A BAR (BlackBerry ARchive, presumably) is a repackaged version of an Android APK, designed to be distributed by BlackBerry and installed on BlackBerry devices. Most of the tooling supplied by BlackBerry surrounds this process of validating that an APK should abide by BlackBerry's requirements and converting that APK into a BAR.

With the release of BlackBerry OS 10.2.1, though, BlackBerry device users can download and install an APK directly. That APK still needs to work within the confines of the BlackBerry Android runtime and must avoid things that will not work there (e.g., replacement home screens). But the user is no longer reliant upon the developer going ahead and creating a BAR file.

At the present time (mid–2014), it appears that the BAR format is still required for distribution by BlackBerry itself through BlackBerry World. Hence, the APK installation option is mostly there for user benefit, for users who install various Android "market" apps and wish to try some of those apps on their BlackBerry devices.

Getting Your Development Environment Established

By and large, developing an app to run on the BlackBerry OS Android runtime is the same as is developing an app to run on any other Android environment. You have your standard choices of IDEs and other development tools, the ability to use third-party libraries, and so forth.

Where things start to differ is in testing, where BlackBerry OS ships a Simulator that fills a role similar to that of the Android emulator. To use the Simulator, you will need to package your app into a BAR file, and BlackBerry provides tools to assist you in that process as well.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Checking and Repackaging Your App

There are two basic technical steps for preparing your app for distribution through the BlackBerry World market.

The first is to validate that your app does indeed stick to APIs that are supported by the BlackBerry Android runtime. This helps prevent apps from appearing on BlackBerry World that are guaranteed to fail.

The second is to convert the APK into a BAR file. BAR files are used for all of the BlackBerry OS runtimes, including ones like Adobe AIR. Your BAR will contain the same stuff that is in your APK, plus some additional metadata, in a format shared by all of the other runtimes, for interpretation and use by the BlackBerry OS.

At the time of this writing (mid–2014), there is a beta edition of an Android Studio plugin available for you to try. In terms of release versions of tools, though, your choices are an Eclipse plugin or command-line tools.

All of these tools are available from the BlackBerry Android developer site.

Eclipse Plugin

BlackBerry publishes an Eclipse plugin that will handle most of the chores for you: test your app for compatibility, convert it into a BAR, apply signing keys, etc. This plugin is certified for use on Windows, OS X, and Linux. This plugin also supports the creation of run configurations to deploy your app to a device or simulator, including supporting IP-based debugging.

Command-Line Tools

The only option available for Linux are the command-line tools (though these also support Windows and OS X).

The main tool is `apk2bar`, which creates a BAR file out of the APK file. It, in turn, will run `apk2barVerifier`, which performs a validation check to see if you obviously use or do something that makes your app incompatible.

To test the BAR, `batchbar-deploy` will upload one or more BAR files to a device or running copy of the Simulator. This works with unsigned and signed BARs.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

To distribute the BAR, you will need to sign it, much like you need to sign an APK to be distributed through major channels. The signing process for BlackBerry OS is a bit complex when compared to the APK signing process.

BlackBerry 10 Simulator

BlackBerry distributes VMWare images that embody a BlackBerry 10 Simulator. You can use these with VMWare Player (Windows and Linux) or VMWare Fusion (OS X), versions 3.1 or higher.

The Simulator fills a role similar to that of the standard Android emulator, allowing you to test your apps for BlackBerry OS without necessarily having a Blackberry OS 10.2+ device, or for testing scenarios that are difficult to test with actual hardware.

In particular, the Simulator offers a wide range of simulated input and output, including:

- Simulated sensor input for the accelerometer and ambient light sensor
- Using a development machine's Bluetooth adapter for testing Bluetooth
- Simulated NFC tags

In addition, the Simulator supports some of the same types of simulated input that you see with the Android emulator, such as simulated GPS fixes and simulated incoming phone calls.

The VMWare image will have its own IP address, which you can obtain from the Simulator running in the image. You can then deploy your BAR to it using the Eclipse plugin or the command-line `blackberry-deploy` tool.

Developing on Hardware

A BlackBerry OS 10.2+ device can run either signed or unsigned BAR files. Unsigned BAR files, though, require a one-time upload of a "debug token", the creation of which requires the same credentials as you would use to sign the BAR in the first place.

How Does Distribution Work?

As with any environment where the Play Store is not available, developers have to determine how best to get their apps to the users of BlackBerry devices.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

As with most non-Play Store ecosystems, there is the official solution... and then there are the other solutions.

BlackBerry World

BlackBerry's own "market" is BlackBerry World. This supports native BlackBerry apps, plus those for runtimes like the Android runtime. So long as your app is packaged as a BAR, it should be able to be released through BlackBerry World, much like how you distribute an APK through the Play Store.

BlackBerry World is a curated marketplace, meaning that BlackBerry staff will review your app submissions and may reject them if they violate requirements or other terms.

Beyond that, BlackBerry World has most of the standard capabilities that you would expect from an app market, such as paid apps, in-app purchases, multiple billing options (PayPal and carrier billing), control for distribution to various countries and mobile carriers, etc.

Alternatives

Starting with BlackBerry OS 10.2.1, you can distribute APK files through your Web site or similar means, and BlackBerry users can download and install them. However, nothing is done to validate that the apps you download will work on BlackBerry OS's Android runtime, as they may use APIs that are not available. IOW, YMMV.

(though, since BlackBerry is a Canadian firm, perhaps that should be YKMV)

Also note that the BlackBerry OS settings have an equivalent to the Android "allow apps from other sources" setting that must be enabled for such installation to occur.

As an example, you can install the F-Droid "market" app to download free and open source Android apps to a BlackBerry OS device. How many of the apps distributed through F-Droid will work on BlackBerry OS, besides F-Droid itself, is unknown.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Device Catalog: Wrist Wearables

A new growth area in mobile technology is the rise of the "wearable": a device designed to be worn on the body in one way or another.

Figure 852: NOTE: Clocks Are Not Normally Considered "Wearables" (image courtesy of Wikimedia Commons)

While large timepieces are not in the domain of what we think of as "wearables", the humble wristwatch has been converted into a new location for computing power. Major manufacturers, such as Samsung and SONY, have launched forays into wrist wearables, and Google has pushed forward with its Android Wear initiative.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In this chapter, we will cover some of the players in the wrist wearable space and what things you may need to think of as you look to start developing apps for such items.

Note that this chapter is focused on general design/implementation issues. Specific coverage of Android Wear, with respect to notifications, can be found in the chapter on advanced notifications.

Prerequisites

Understanding this chapter requires that you have read the core chapters of the book.

Divvying Up the Wearables Space

For the purposes of this chapter, we will consider wearables along two axes:

- What OS does the wearable run?
- What OS can we use for developing apps for the wearable?

Devices vs. Accessories

For this chapter, a "device" is a wearable that runs a recognized mobile operating system. Mostly, right now, that is Android, for devices like the Android Wear watch series, the Omate TrueSmart, and the I'm Watch. In principle, in the future, other operating systems used on phones or tablets could make the jump to run directly on the wearable.

In contrast, an "accessory" is a wearable that runs... something else. Usually, this is some proprietary OS, one we do not work with directly. So, the Pebble smartwatch is an accessory, as the Pebble runs its own OS, not something like Android that is popularly used on phones and tablets.

App OS

In the case of devices, usually we will write apps for the OS of the device. Writing an app for the Omate TrueSmart is reminiscent of writing an app for a small-screen Android phone, for example. Writing apps for Android Wear are a bit different, in

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

that Google has a specific way to package those as part of an app for a mobile device, but otherwise it is relatively similar.

In the case of accessories, the accessory maker may have provided SDKs to allow for apps to be created that run on a phone or tablet that the accessory is tethered to. Basically, the phone or tablet provides the "brains", while the accessory is simply a very tiny input/output terminal. Since this book is on Android application development, either an accessory supports apps that run on Android, or it does not (e.g., it only supports iOS at the moment).

NOTA (None Of The Above)

Some devices do not fit these buckets, for the simple reason that they do not support app development of any kind at present. For example, the first-generation Galaxy Gear from Samsung, and the Qualcomm Toq, are closed environments. In the case of the Toq, it is unclear if anyone has an SDK for it. In the case of the Galaxy Gear, Samsung appears to have a private SDK that they have made available to select partners, eschewing the broader Android development ecosystem for the time being.

Example Wrist Wearables

The following sections outline several wrist wearable devices. This is not meant to be an exhaustive list. Instead, it is here to give you a sense of what is available, both in terms of brands and opportunities for app developers.

Android Wear

Google announced the Android Wear initiative at Google I|O 2014, and the first devices soon followed. Some are square (e.g., Samsung Galaxy Gear), while others are round (e.g., Moto 360). All are wearable devices, as they run Android 4.4W, which is standard Android minus a few things, like `WebView`.

Fitbit

Fitbit is a wrist wearable, one with only a few LEDs for output, designed almost exclusively for collecting sensor input (e.g., pedometer). While the Fitbit site offers an API, it is not an API to the Fitbit wearable, but rather to a Fitbit Web service, for retrieving Fitbit results. As such, the Fitbit does not qualify as either a device or an accessory in terms of this chapter's model.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

I'm Watch

The I'm Watch was one of the first wrist wearable devices, running Android. The I'm Watch runs Android 1.6, which works well in limited hardware, and with only 128MB of RAM and a ~400MHz CPU, the I'm Watch hardware is limited. From the standpoint of Android developers, Android 1.6 is "rather long in the tooth", as most developers have abandoned Android 1.x development, and many have even stopped worrying about Android 2.x.

The I'm Watch offers a 1.5", 240x240 pixel screen, 4GB of on-board storage, and a couple of sensors (accelerometer and magnetic field). Distribution of apps for the I'm Watch is handled through the manufacturer's own storefront, though developers can deploy apps through `adb`.

MetaWatch

MetaWatch was born out of a Kickstarter project. It is an accessory, running some tiny OS of its own, designed for "widgets" (MetaWatch apps) to be powered by a tethered device. Alas, momentum seems to have stalled on their app SDKs, and so there does not appear to be a well-supported way to create apps to run on Android devices that push content to the MetaWatch.

Omate TrueSmart

Another Kickstarter was for the Omate TrueSmart. This is a wearable device, powered by Android 4.2. Writing apps for the Omate TrueSmart closely resembles developing apps for a tiny Android phone.

The hardware specs are more in line with modern devices, when compared with the I'm Watch:

- Dual-core ARM 1.3GHz CPU
- 512MB or 1GB of system RAM
- 4–8GB of internal storage, plus microSD expansion
- 1.5" 240x240 touchscreen
- WiFi, mobile data, and Bluetooth

As of February 2014, the Omate TrueSmart has shipped to most of its Kickstarter backers and should be available for sale to others shortly.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Pebble

Of course, the Kickstarter that, um, kick-started the whole "Kickstart-a-wearable" phenomenon was the Pebble smartwatch. The Pebble is an accessory, where the on-board OS runs apps written in C that can optionally communicate with Android or iOS apps running on tethered devices.

Qualcomm Toq

Qualcomm, in late 2013, decided to get into the wearables game directly, via their Toq smartwatch. This is an accessory, designed to work in conjunction with an Android device. In February 2014, Qualcomm released an SDK to allow Android developers to create apps that can work with the Toq.

However, the license terms for this SDK contain clauses that should be discussed with qualified legal counsel, including:

- granting Qualcomm the right to modify and demonstrate your application to whoever it wants to;
- precluding the development of apps (or libraries) that employ a number of popular open source licenses;
- prohibiting distribution of an app without Qualcomm's prior written permission; and
- giving Qualcomm the right to tell you to uninstall your app, from any device, at any time, for any reason

Samsung Gear Series

One of the bigger brands to get into wrist wearables has been Samsung, with their Galaxy Gear device. The Galaxy Gear is a device, running Android. However, as of February 2014, there is no publicly-available SDK for writing apps for this device, nor any officially-supported means of installing third-party apps on the device. You can load apps on it via `adb`, though this is not supported and has substantial limitations (e.g., no Internet access).

In February 2014, Samsung announced the Gear 2 and Gear 2 Neo. These are devices, but running Tizen, not Android. Samsung released an SDK for these devices](http://developer.samsung.com/samsung-gear) in March 2014.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In February 2014, Samsung also announced the Gear Fit. This *does* have an SDK (as part of Samsung's overall mobile SDK package), and qualifies as an accessory.

SONY SmartWatch and SmartWatch 2

SONY is another major brand to get into wrist wearables. In fact, they were the first major brand to do so, debuting their SONY SmartWatch back in 2012. SONY then released the SmartWatch 2 in 2013. Both are accessories, where the apps for the SmartWatch actually run on a tethered Android device, with the SmartWatch serving as a display surface and input mechanism (via simple taps and gestures on the touchscreen).

That Theorized Apple Watch

Of course, the brand that everyone is waiting for to get into the wearables space is Apple. There have been rumors of an "iWatch" for quite some time, though with no actual device having been released as of February 2014.

WIMM One (RIP)

There used to be a firm called WIMM Labs, which made a wrist wearable device called the WIMM One. The device was much along the lines of the I'm Watch and Omate TrueSmart, falling roughly in between them in terms of Android OS version and hardware capability. Their approach was to be more of an OEM, though, looking to supply branded versions of the WIMM One to firms who might want a wearable but were not in that technology space, such as clothiers like Nike.

Google acquired WIMM Labs, at which time the WIMM One was pulled from the market. Some of the WIMM Labs capabilities perhaps moved into the Google Glass project, though it is certainly possible that Google could offer a "Nexus Watch" sometime in the future.

Strategic Considerations

Before leaping into extending your app to support wearables of any sort — ones worn on the wrist or otherwise — you really need to spend some time thinking about whether or not this is the best thing for you and your users.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Do We Bother?

While wearables are touted as “the next big thing”, there have been plenty of such “things” that have never gotten to be that big. This is not to suggest that wearables will go the way of tree-style Web site directories, but they may not become quite as popular as the phones and tablets they are meant to accompany.

While developing for wearables has a “cool factor”, the time you spend on wearables is time taken away from other things. For developers, you lose time that you might have spent on other features of your app. For independent developers, the time might come out of your time budgets for marketing or product research.

While a pure hobbyist might pursue wearables just because it is an interesting area to get into, many other developers need to weigh whether adding wearable support will be worth the time investment.

Which Types?

Developing apps for Android wearable devices, like the Omate TrueSmart and the I’m Watch, is comparatively straight-forward. Device limitations may cause you to spend some more development time, such as crafting a UI that works well on ~1.5" displays. And, depending on the device, you may have to use alternative APIs for some basic operations — the WIMM One, for example, did not have its own WiFi or mobile data, and so you had to wait until the WIMM One told you it was safe to try Internet operations, when the device was tethered to a phone that had its own connectivity. But, at the end of the day, it is still the same Android development that you have come to love (or at least tolerate, occasionally through gritted teeth).

Accessories, or devices running other mobile operating systems, require custom development. And, since there are no standards in this area yet, it usually requires per-device custom development. This may be a graceful extension of logic you already have, such as repurposing your app widget code to use for a watch. But the APIs for delivering content to, say, a SONY SmartWatch do not closely resemble app widget APIs, which will require some restructuring of your code, in addition to the SONY-specific work. How little or how much work this is will vary by app and, secondarily, by wearable.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Which Devices?

Once again, developing for Android wearable devices may be a small enough leap that you could consider them as a matter of course. This is especially true of "pure" enough devices that your business logic will just work, with only a watch-sized UI being needed. It may be that one pulse of work can get your app working on multiple Android wearables that share common capabilities (similar screen sizes, standard connectivity, etc.).

However, beyond that, you are going to need to consider whether developing for a specific wearable is worth the specific work. For example, many of these wearables use their own "market" or "store" for app distribution. Will listing your app in their market get you enough results to make the work worthwhile? Will the wearable maker offer any sorts of co-marketing opportunities, to help promote your app? Does their market offer things that you are used to from more traditional markets, like in-app purchases, that your app uses? Is your app so tightly tied to the Play Services Framework (e.g., Maps V2, GCM, `LocationClient`) that it simply cannot be distributed by any other means?

So, for example, the I'm Watch has their own store, and you will be limited by the capabilities of that store. SONY, on the other hand, expects apps for their SmartWatch line to be distributed however you want, particularly through the Play Store, which gives you more options.

Tactical Considerations

Given that, for strategic reasons or "just because", you have decided to go ahead and work on extending your app to wearables, there are a variety of things you are going to need to think about as you start in on that work. A lot of the details will vary by the specific wearable device or accessory you are trying to support, but some common themes will emerge. Some of those themes are discussed in this section.

The Postage Stamp User Interface

When writing apps to display to televisions — whether for a dedicated box like an OUYA game console or for an external display attached to a traditional touchscreen device — you think of the "10-foot user interface". Here, "10-foot" is not referring to the size of the television (usually), but the distance between the viewer's eyes and the television. Even though you may have a nice 1080p television, you sit so much farther back from it than you do a 1080p phone or tablet, that the UI needs to take

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

this into account. Also, input options for a television tend to be different than on a phone or tablet, as televisions rarely are touchscreens.

Writing apps for wearables has a similar UI distinction, though in a much smaller dimension. Trying to determine how best to package your content to be accessible on a ~1.5" screen is tricky.

One approach is to consider your app widget, if you have one. An app widget on a phone or tablet often times takes up a watch-sized chunk of space on the home screen. Also, app widgets have constrained forms of user input (simple taps, scrolling or swiping only on select widgets like `ListView`), matching the constraints imposed by some wearable APIs and by the reality of large fingers and small touchscreens.

Your UI design for your app widget may be a fine starting point for your UI design for your wrist wearable. In some cases, the same UI will work, while in other cases, some tweaks may be required:

- You designed your app widget to be rectangular, rather than square, so a slight redesign will be needed to take best advantage of a (roughly) square smartwatch display
- You designed your app widget with `ListView` at its core, but the particular wearable you are targeting does not support `ListView`, so you have to revert to other UI structures
- You designed your app widget around a rich color screen, but the wearable that you are supporting offers a more limited color palette, due to its use of particular types of display technology (e.g., “e-ink”)
- And so on

Connectivity (Or Lack Thereof)

Some wearables have their own WiFi radios. Some have their own mobile data options, if they are outfitted with a suitable SIM card. If these are devices, writing an app for the device’s operating system should allow you to connect to the Internet more or less as you do with an app for a phone or tablet.

Some wearables have apps that run on the device, but the device lacks any native Internet connectivity. In that case, you may have access to some APIs that allow you to work with an app on a tethered phone or tablet, and be able to access the Internet that way.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Wearables where the app itself runs on the phone or tablet return you to regular connectivity options, as the wearable does not need independent Internet access.

In sum, how well you can get to the Internet is partly determined by the nature of the wearable, and partly determined by normal mobile development practices (e.g., are we connected to WiFi?).

Power (Or Lack Thereof)

The Omate TrueSmart is one of the larger wrist wearable devices. It has a 600mAH battery, a fraction of the size of batteries in most Android phones, let alone tablets.

On the one hand, a wearable will tend to draw less power, simply because the wearable manufacturer will have taken steps to minimize power drain. One example is turning off the LCD more aggressively than you might expect to see on a corresponding phone or tablet.

That being said, power drain is an issue on phones and tablets already; it will be that much worse in the world of wearables. In some cases, you have *two* batteries' worth of power to consider: the wearable's own battery, plus the battery of some phone or tablet (particularly in the case of an accessory, where the app runs on that phone or tablet).

Security (Or… Well, You Get the Picture)

Few, if any, wearable devices offer much in the way of on-board security, such as full-disk encryption, or even a PIN or password for access. That is because a wearable is designed for rapid access, and such security usually impedes such access. The assumption is that if the wearable falls into the wrong hands, that anything on the wearable is subject to inspection.

As such, you need to be a bit careful about what you use the wearable for. Most apps involving a wearable may have limited security requirements, but not all. Wearable devices, in particular, have the risk that they offer some amount of on-board storage that your on-device app can use, making it somewhat more likely that you might try actually storing something there... something that the user might want to keep secret.

Similarly, the communications between a wearable and a tethered phone or tablet may be insufficiently secure. For example, Bluetooth communications may be secure, or they may not, depending on hardware and circumstances.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

This does not prohibit the creation of secure wearable apps, but it make it more difficult.

What About Android Wear?

Android Wear is an initiative by Google for wearable devices, with a particular emphasis on wrist wearables. This book will cover Android Wear in a future edition.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Device Catalog: Android TV

Google not only offers the Chromecast, but also offers Android TV as a way to get content "into the living room". Android TV devices — whether they be set-top boxes or are integrated into televisions directly — run Android apps directly, unlike Chromecast. Android TV, therefore, is a competitor to devices like Amazon's Fire TV.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book. Having read the chapter on "ten-foot" user experiences is also a good idea.

Hey, Wait a Minute… I Thought the Name Was "Google TV"?

You can be forgiven for any confusion over the names.

Google TV was Google's initial attempt to get content onto televisions. Debuting in 2011, Google TV has some of the same characteristics as does Android TV:

- It ran Android apps directly (at the time, on Android 3.1)
- It could be a dedicated set-top box, integrated into other sorts of media players (e.g., Blu-Ray), or integrated into televisions

However, Google TV did not prove all that popular.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

In 2014, Google announced that they were no longer supporting app development for Google TV, much to the consternation of the ~17 people still using Google TV devices.

That being said, designing an app for Android TV resembles designing an app for Google TV or for any other "ten-foot" user experience. Hence, design guidance that you may run across for Google TV may have some tips that are still relevant for Android TV and other TV-centric Android environments.

Some Android TV Hardware

Android TV debuted in 2014. However, as of the end of 2014, there were a total of two Android TV device models... both available from Google. While other manufacturers had announced plans regarding Android TV, none were available at the time of this writing.

ADT–1

When Google TV was announced, there was a similar lack of hardware. To help ensure that there were TV-capable apps at the time of a wider Google TV rollout, Google offered free developer devices to various firms and solo developers.

Google did the same thing with Android TV, where they offered a free ADT–1 device to qualified registrants. While deliveries of the ADT–1 were spread out over a few months, they generally arrived significantly in advance of production Android TV hardware.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 853: ADT–1 Developer Android TV Device

Nexus Player

The first production-grade Android TV device is Google's own Nexus Player. As with the ADT–1, the Nexus Player has an HDMI port for connecting to a TV (or projector, monitor, etc.). It also has its own remote control and power adapter.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 854: Nexus Player Android TV Device

Both the ADT–1 and the Nexus Player are presently running Android 5.0.

What Features and Configurations Does It Use?

Android has built into the SDK a fair bit of device flexibility. Most of this comes in the form of configurations (things that affect resources) and features (other stuff). If your application can handle a range of configurations and features, or can advertise that they need certain configurations or features, they can handle Android TV or arrange to not be available for Android TV on the Play Store.

Screen Size and Density

Android TV devices are always categorized as `xlarge` screen size.

Densities, however, are a bit more complicated.

Android TV is for use with HDTV, whether Android TV is integrated into the television or it comes as an external set-top box. There are two predominant HDTV resolutions, known as 720p (1280x720) and 1080p (1920x1080). A 1080p television

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

will be categorized as an `xhdpi` density device. A 720p television will be categorized as a `tvdpi` device. `tvdpi` is for devices around 213dpi, in between `mdpi` and `hdpi`. In practice, you might elect to skip `tvdpi` for your drawable resources, allowing Android to resample your `mdpi`, `hdpi`, or `xhdpi` drawables as needed.

Input Devices

Android TV will not normally be navigated using a touchscreen. Instead, the normal form of input will be a D-pad remote. Developers will need to ensure that their apps are navigable this way.

Other Hardware

Android TV has no sensors, no camera, no microphone, and no telephony features. As such, any application requiring such features will not run on Android TV and will not even show up in the Play Store for such devices.

Bear in mind that some of these will be driven by permissions. If you ask for the `SEND_SMS` permission, Android will assume you need android.hardware.telephony unless you specifically state otherwise, via a `<uses-feature>` element for `android.hardware.telephony` with `android:required="false"`.

What Is Really Different?

Beyond the features and configurations, there are other things about Android TV that will depart from what you might expect for an Android environment, due to the nature of the TV set-top box platform and the Android implementation upon it.

Overscan

Since Android TV typically uses a television for its primary output, overscan can be an issue. Addressing overscan is covered as part of the chapter on the “10 foot UI”.

Ethernet

While Android TV devices will generally be connected to the Internet, it may not be via WiFi. Since Android TV devices generally are not portable, some will have Ethernet jacks, and hence some users will elect to wire in their Android TV as opposed to using WiFi.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

The upshot is that you should not assume that `WifiManager` will necessarily give you useful results. Also, `ConnectivityManager` should report wired Ethernet as `TYPE_ETHERNET`, added in API Level 13, when you call methods like `getActiveNetworkInfo()`.

Location

Generally speaking, Android TV devices will tend not to move, earthquakes and large dogs notwithstanding.

As such, Android TV devices do not have GPS receivers. Rather, location is determined in an approximate fashion via address-based lookups, using a postal code. Hence, asking Android for a GPS fix on a Android TV device will be ineffective.

However, since users of Android TV devices tend not to be moving much at the time, it is a bit more likely than normal that they will want information about some location other than where they are. If your app is exclusively tied to providing information about their current location, you may wish to consider how you could extend your app to help users get information about other places that they may be interested in.

Media Keys

Android TV devices are usually manipulated by remote controls. Some of these remotes will have lots of buttons, such as media-specific buttons for play, pause, etc.

The `KeyEvent` class has had support for some media buttons since API Level 3, mostly for use with wired headsets. API Level 11 added a bunch more media buttons. Your Android TV application may wish to respond to these, via `onKeyDown()` in a `View` or `Activity`.

In particular, an Android TV application should not be using on-screen controls for play, pause, etc., as they take up screen space that probably could be put to better use. Rather, use layouts that offer such controls for touchscreen devices (e.g., phones and tablets) but rely on the media buttons for non-touchscreen devices.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Getting Your Development Environment Established

Android TV emulator images are available in the SDK Manager for Android 5.0 and above:

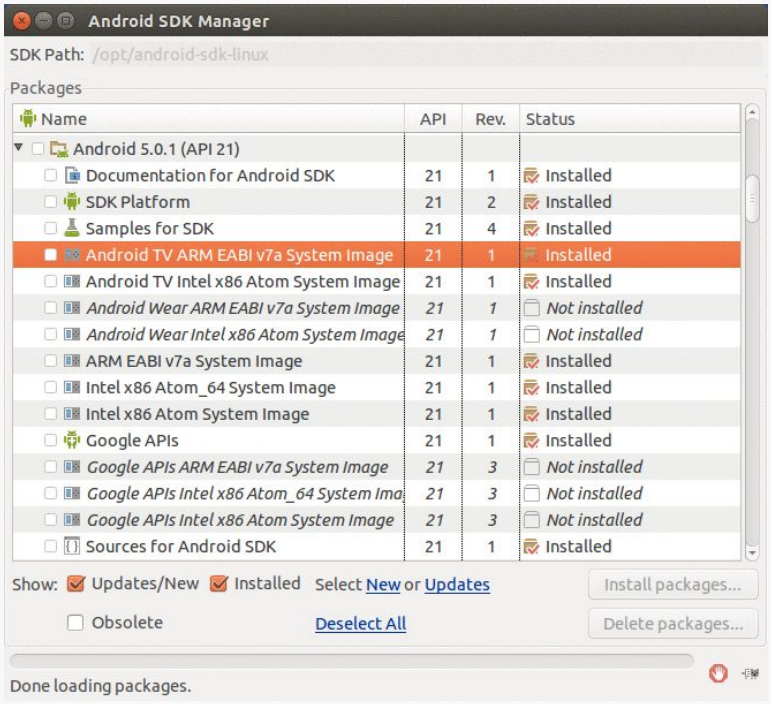


Figure 855: Android TV Emulator Images in SDK Manager

Your AVD Manager can then help you set up TV emulator images, for rather large theoretical screen sizes:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

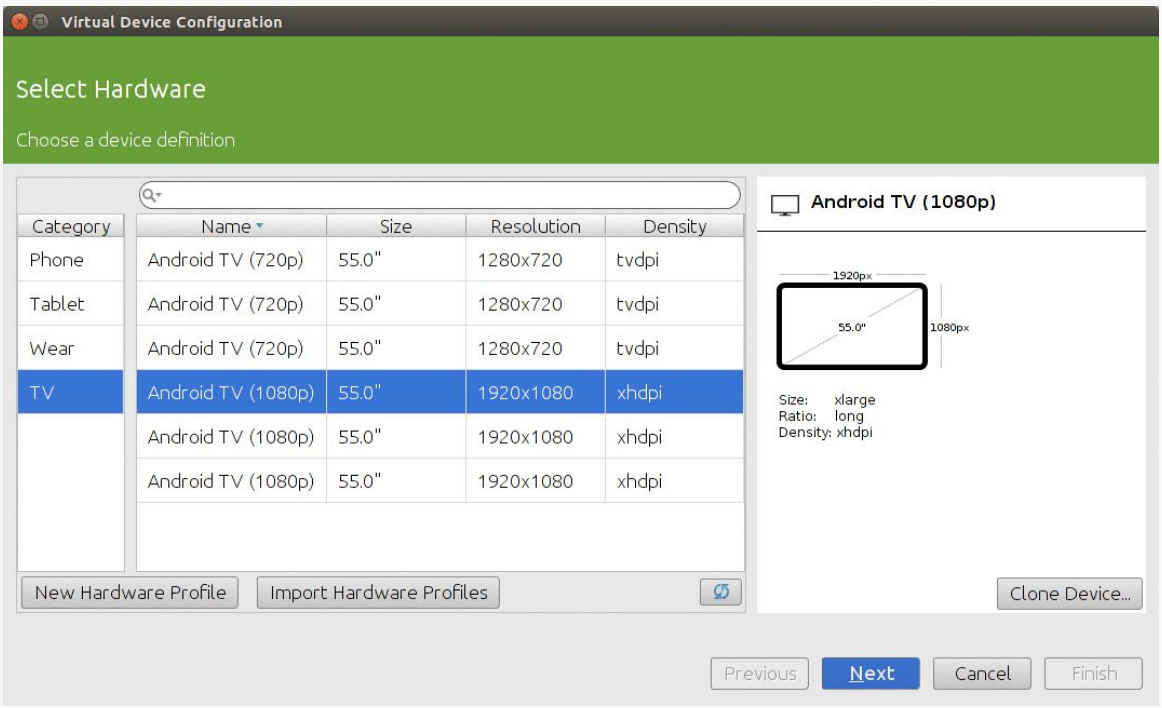


Figure 856: Android TV Hardware Profiles in AVD Manager

The rest of the emulator setup is the same as you would have for phone or tablet emulators.

The emulator even has "leanback" UI characteristics, rather than a standard Android emulator home screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

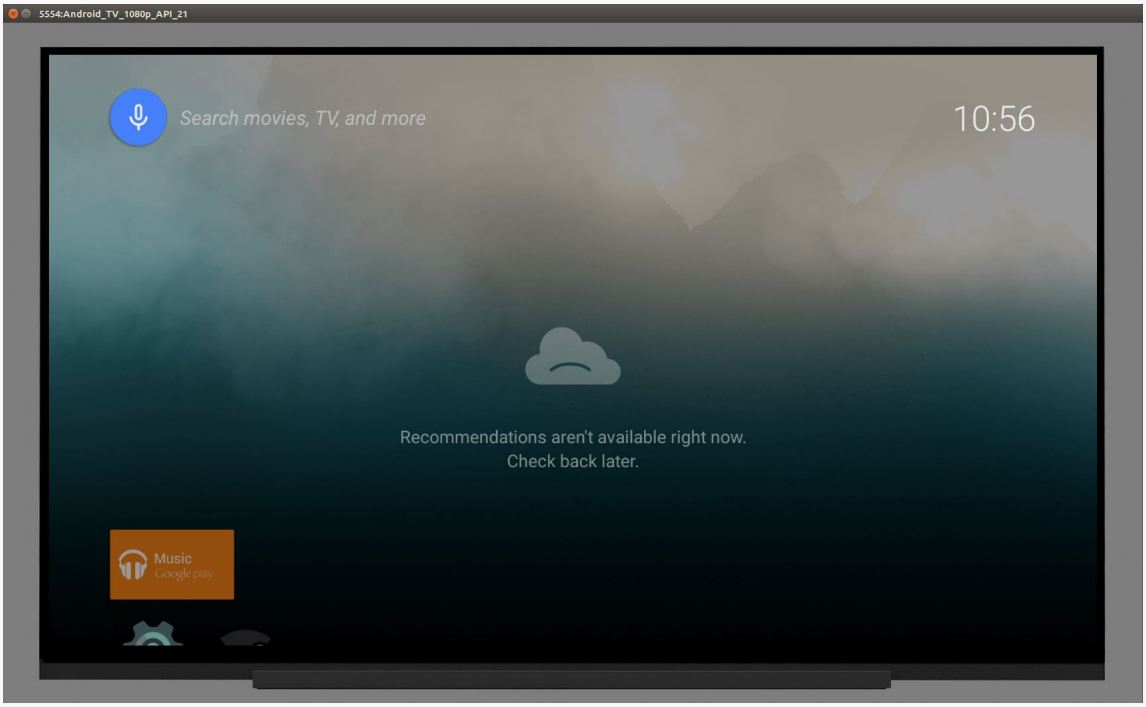


Figure 857: Android TV 1080p Emulator, As Initially Launched

Connecting to Physical Devices

The ADT–1 and Nexus Player have micro-USB ports for debugging purposes. This, though, requires that your player and your development machine be with USB cable reach of one another. The author of this book uses a pico projector to be able to work with TV-native devices (Android TV, Fire TV, etc.) or external displays (HDMI, MHL, etc.).

To enable an Android TV for debugging, you will need to enable developer options, much like you do for a mobile device (click on the build number 7 times in the Setting's About screen). Then, in the "Developer Options" screen, you can go into Debugging and elect to enable "USB debugging".

How Does Distribution Work?

Your app probably falls in one of three buckets: you want it on Android TV (along with other devices), it *only* supports Android TV, or it will not work on Android TV. Whichever of those buckets best fits your device will determine the manifest

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

settings you will want to ensure that the Play Store (and perhaps other third-party markets in the future) will honor your request.

Getting Your App on Android TV

The first criterion for getting your app visible to Android TV devices on the Play Store is to add a `<uses-feature>` element to your manifest, indicating that you do not require the `android.hardware.touchscreen` feature:

```
<uses-feature android:name="android.hardware.touchscreen"
android:required="false"/>
```

By default, Android assumes that you need a touchscreen, and so without this clarification in your manifest, you will not appear in the Play Store.

Also, add similar `<uses-feature>` elements for any hardware that you might like to use where available but do not absolutely need, particularly hardware that Android TV may lack. The documentation outlines the features that Android TV devices will likely lack.

You need to have an activity that has an `<intent-filter>` for `ACTION_MAIN` and `CATEGORY_LEANBACK_LAUNCHER`. This will be your launcher activity on Android TV devices, instead of your `ACTION_MAIN`/`CATEGORY_HOME` activity. Of course, you are welcome to have one activity serving as the launcher for both types of devices, if you can come up with one with a solid presentation for both mobile devices and TVs.

In addition, do not have any activities with `android:screenOrientation` set to `portrait`, as Android TV devices always display in landscape

Supporting Only Android TV

If your app only supports Android TV, in addition to the above requirements, you should also add one more `<uses-feature>` element to your manifest:

```
<uses-feature android:name="android.hardware.type.television"
android:required="true"/>
```

This will filter you out of the Market for all non-TV environments.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Avoiding Android TV

If your app specifically is untested on Android TV, you need to have something in the manifest that will keep you *off* Android TV devices' views of the Play Store. The easiest is to say that you need a touchscreen:

```
<uses-feature android:name="android.hardware.touchscreen"
android:required="true"/>
```

Note that `true` is the default setting for this particular feature, though putting it in your manifest to remind you that you do require a touchscreen is a good idea.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Device Catalog: Amazon Fire TV and Fire TV Stick

Amazon has joined the TV set-top box fray with the Fire TV and, more recently, the Fire TV Stick. These devices are powered by FireOS, Amazon's variation of Android. And, as with other Amazon FireOS devices, like the Kindle Fire tablet series, you can write apps that run on Fire TV and the Fire TV Stick.

This chapter will review these devices from a developer's standpoint, to help you create apps for this platform. Note that the Fire TV line of devices have some elements in common with Amazon's Kindle Fire series of tablets, covered elsewhere in this book.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, along with the chapter on the Kindle Fire tablets. Reading the chapter on the ten-foot user interface is also recommended, either before or after this chapter.

Also, some sections will make reference to Android TV as a point of reference.

Introducing the Fire TV Devices

As of January 2015, there were two major flavors of Fire TV device: the original Fire TV, and the Fire TV Stick.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Fire TV

As with most Android-powered set-top boxes, the original Fire TV is small and is designed to connect with your television (or monitor, or projector, or whatever) via HDMI:

Figure 858: Fire TV

It comes with a small wireless remote designed for basic controls, akin to what you might find on other streaming boxes or similar entertainment devices:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 859: Fire TV Remote

Optionally, you can get a gaming controller that works with the Fire TV. While the regular controller is fine for navigating the Fire TV UI, the gaming controller will be more suitable for more serious game play:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 860: Fire TV Gaming Controller

While the Fire TV is powered by Android, the on-device UI is definitely targeting a set-top box environment. The home screen is dominated by media, coming from whatever supported streaming content providers you have set up with the Fire TV (e.g., Amazon Prime):

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

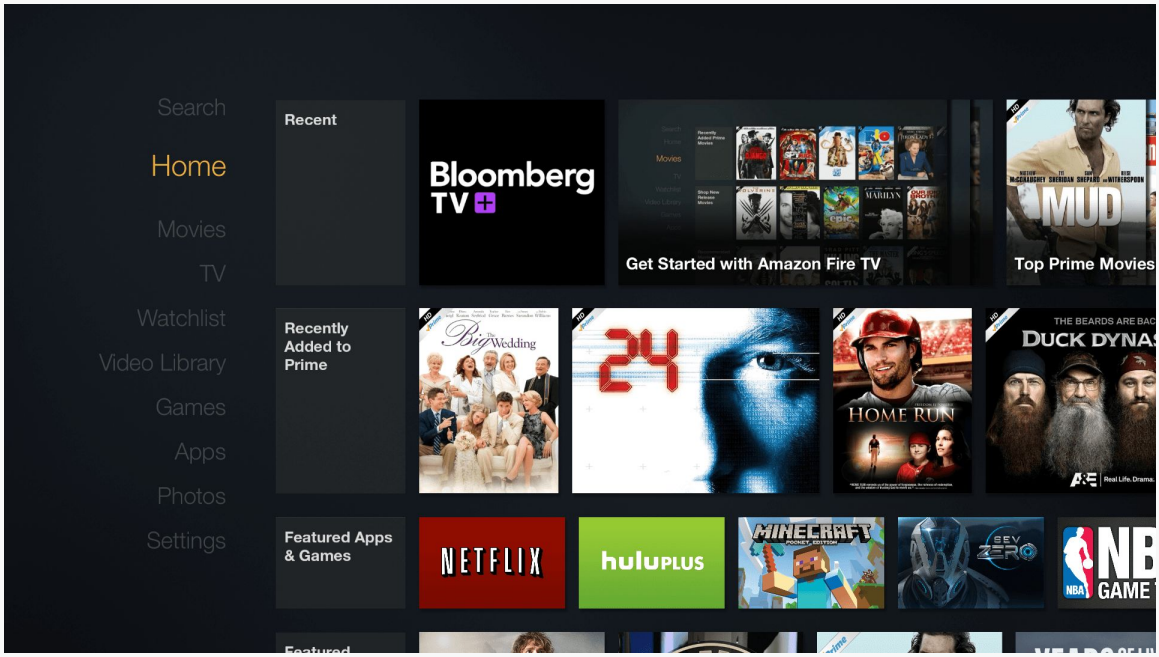


Figure 861: Fire TV Home Screen

The “Apps” section shows a mix of what is installed and what is available for you to download, with “cloud” icons indicating apps that are available but are not presently installed:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

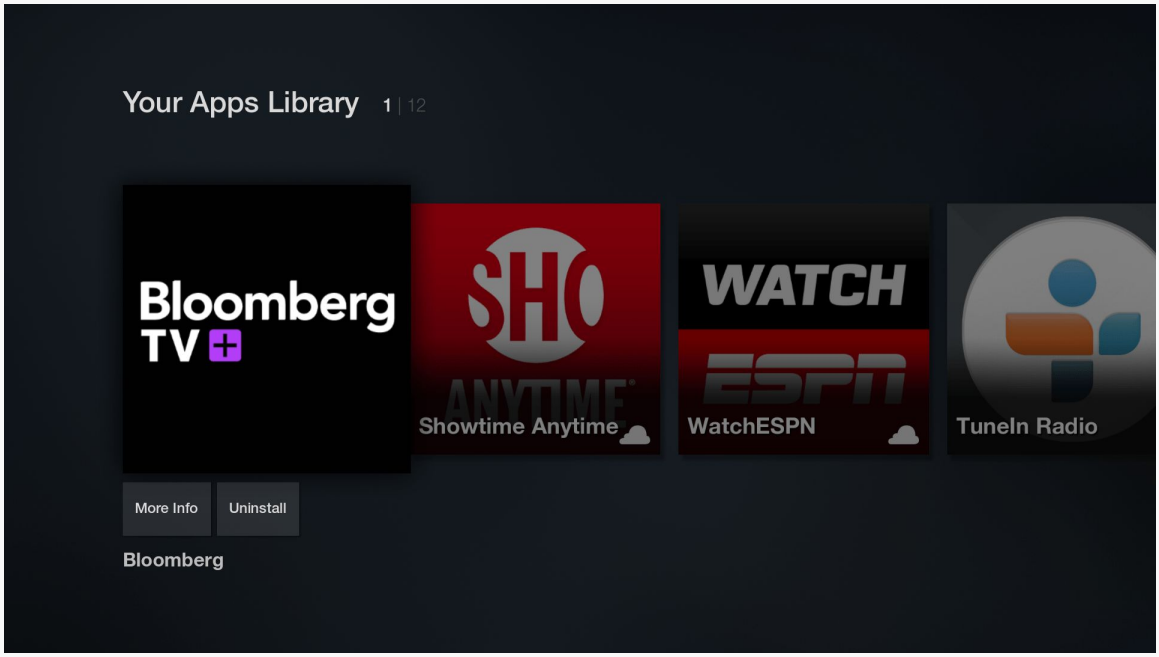


Figure 862: Fire TV "Your Apps Library"

FIre TV Stick

The Fire TV Stick is physically substantially smaller than is the Fire TV, designed to more closely resemble the Chromecast:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 863: Fire TV Stick, with Remote, AC Adapter, and HDMI Extension Cable

The user experience of a Fire TV Stick, though, is largely the same as with a regular Fire TV. Both run the same FireOS environment, with the same sort of browsing metaphor.

The biggest difference is that the Fire TV Stick is less powerful (less RAM, weaker CPU and GPU). For conventional apps, this is unlikely to be a problem, as the Fire TV Stick is as powerful as many standard Android phones and tablets. Games, however, are more likely to stress the hardware.

From the user's standpoint, the Fire TV Stick's big selling point is its low price, about a third of what the Fire TV costs.

What Features and Configurations Do They Use?

The Fire TV device family behaves a bit like a "mashup" of other TV-centric devices (Android TV, OUYA, etc.) and the Kindle Fire series.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

OS Version

At the present time, Fire TV devices are powered by a version of FireOS that corresponds to Android 4.2.2 (API Level 17). It is possible that Fire TV devices could be upgraded to a newer version of FireOS in the future, one based on a newer version of Android, but there are no guarantees.

Screen Size, Density, and Orientation

One area where the Fire TV series differs from pretty much anything that came before it comes with the behavior of the screen. As with other TV-centric devices like Android TV and the OUYA, the assumption is that the screen is locked to landscape. However, beyond that, Amazon has struck out on its own in terms of screen characteristics.

With Android TV, the screen size and density are based on the type of display that the Android TV box is plugged into.

With the Fire TV devices, your code deals with a rendering surface of 1920x1080 pixels, regardless of whether the device is plugged into a 1080p screen or something else (720p, 4K, etc.). The density is always treated as `-xhdpi`, giving you a resolution of 960dp by 540dp, and a `-large` screen size. The hardware will handle scaling the output down (or, in theory, up for 4K) as needed.

On the plus side, this simplifies app development, as you do not need to deal with different screen capabilities yourself. However, it remains to be seen how well the Fire TV will scale the output.

Also, note that Fire TV devices do not address overscan when they apply this scaling. Hence, you will want to keep your content away from the edges of the available space, only showing your background there, in case those edges are not visible on some televisions.

As with any TV-centric device, since the screen is locked to landscape, activities that are locked to portrait (e.g., `android:screenOrientation="portrait"` in the manifest for the activity) will behave oddly. If you are targeting devices like the Fire TV series, be sure to allow your activities to work in portrait *or* landscape, not just portrait.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Input Devices

As noted earlier, Fire TV devices ship with a simple remote control, designed for media management (play, pause, etc.) and basic D-pad navigation. Some users may also elect to pick up a Fire TV game controller or use other Bluetooth game controllers. Technical details for working with these input devices can be found later in this chapter.

Conversely, Fire TV devices do not have touchscreens. You cannot assume that the user can tap on random portions of the screen. Instead, you need to make sure that your app is completely reachable via a D-pad. This is described in greater detail in the chapter on the "ten-foot UI".

Hardware Features

As a set-top box, Fire TV devices lack a lot of hardware that you normally associate with phones and tablets, such as:

- GPS
- camera
- microphone
- sensors
- telephony

The microphone is actually a bit complicated. The Fire TV remote has a microphone and a button to activate it. The Fire TV Stick remote does not have a microphone. There is a Fire TV Remote app available on the Play Store and Amazon Appstore for Android that offers microphone input as well. However, all these microphones are just for Fire TV voice search. There is no means for developers to use these for arbitrary audio input, the way that a microphone on a phone or a tablet can.

What Is Really Different?

Some things are "really different" simply because the output screen is TV-sized, not phone-sized or tablet-sized. See the chapter on the "ten-foot UI" for more about this.

Some things are "really different" in the same way that they are "really different" for the Kindle Fire series, such as having nothing "Googly", like Google Maps or Google

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Cloud Messaging. Note that the Amazon Maps API does not work on the Fire TV series, as those devices have no location services.

The biggest additional "really different" item is the limitation on where apps can come from, which is discussed later in this chapter as part of the overall app distribution options for Fire TV devices.

Note that Fire TV devices have their own way of representing notifications. A standard Android `Notification` will not appear anywhere on a Fire TV device. Instead, you will need to support the Fire TV series' own notifications API. While their API is modeled after Android's `Notification` and `NotificationManager`, they do have their own classes, and the results are more reminiscent of `Toasts` and `Dialogs`.

Also note that Fire TV devices do not display an Android action bar, representing those items in a pop-up menu triggered by the MENU button of the Fire TV remote control.

Casting and Fire TV

The success of Chromecast means that TV-connected Android devices will tend to be compared to Chromecast, particularly in terms of how apps on phones and tablets can work with the TV-connected device.

At the time of this writing, Amazon has not shipped anything for Fire TV that would enable apps using `MediaRouteActionProvider` and `RemotePlaybackClient` to work with Fire TV. In principle, Amazon or somebody else could implement a `MediaRouteProvider` and distribute that as an app that goes on ordinary Android phones and tablets, where that `MediaRouteProvider` enables media routes pointing to a connected Fire TV. The `MediaRouteProvider` would forward requests from the phone or tablet to the Fire TV, which would be displayed by some app on the Fire TV.

Amazon more formally supports the "Discovery and Launch" (DIAL) protocol with Fire TV. DIAL allows an app on a phone or tablet to launch apps on Fire TV. Unfortunately, documentation for this is lacking at the present time.

Amazon claims that the Fire TV series supports serving as a Miracast endpoint, so phones and tablets could connect to Fire TV devices and use them as an external

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

display. Reports from users suggests that this rarely works, and Android developers would be wise to not rely heavily upon it at this time.

Getting Your Development Environment Established

Developing for the Fire TV series, as with developing for the Kindle Fire series, is best accomplished using an actual Fire TV or Fire TV Stick device. That being said, having an emulator that at least resembles a Fire TV device may be useful for debugging purposes.

Emulator Configuration

Amazon is distributing an SDK add-on that supplies an emulator image you can use, in theory, to emulate a Fire TV device.

To install it:

- Start the SDK Manager, such as via the toolbar button in Android Studio or Eclipse
- Choose Tools > Manage Add-on Sites from the SDK Manager main menu
- Click on the User Defined Sites tab, and click the New... button
- Fill in `https://s3.amazonaws.com/android-sdk-manager/redist/addon.xml` as the URL in the field in the "Add Add-on Site URL" dialog, then click OK (to close up that dialog), then click Close (to close up the Manage Add-on Sites dialog)
- Wait for the progress bar at the bottom of the SDK Manager to finish
- Install the Fire TV entry in API Level 17

This will give you an "Amazon Fire TV SDK" option as an API level target when setting up an emulator image. However, you will also need to set up a device definition that emulates a suitable screen resolution (e.g., 1080p). And, the resulting emulator does not even vaguely resemble a Fire TV device in terms of the firmware and such.

In truth, at present, you will need to use Fire TV hardware to develop for Fire TV.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Developing on Hardware

The primary thing notably different about testing your app on a Fire TV device is that it does not use a micro USB cable for the `adb` connection, the way that you may be used to from testing with most Android hardware. It does not even use some sort of proprietary cable. Instead, `adb` runs over the network.

To set this up, you must first enable USB debugging, much like you do with other Android devices. On a Fire TV device, this is in Settings > System > Developer Options:

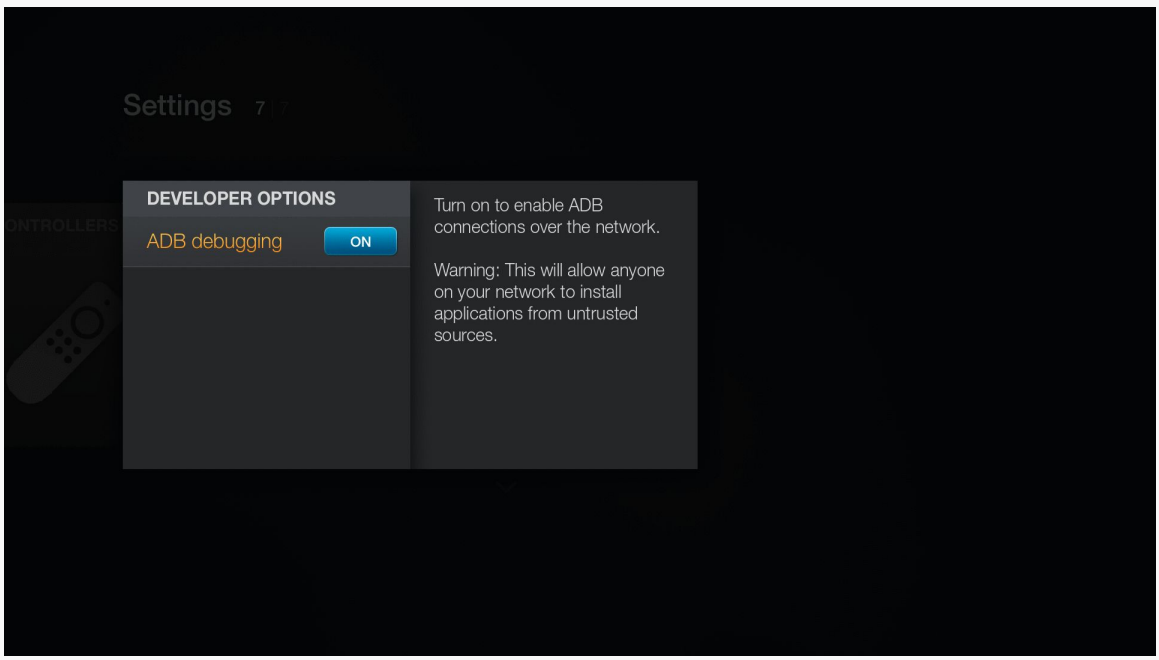


Figure 864: Fire TV Developer Settings

You will then need the IP address of your Fire TV device. Most likely the easiest way for you to find that is via the Settings > System > About > Network screen:

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Figure 865: Fire TV Network Settings

You can then run `adb connect` (followed by the IP address) at a command prompt to make the connection. You may need to stop and restart `adb` before doing this, via `adb kill-server` and `adb start-server`.

At this point, if all is set up properly, `adb devices` will list the Fire TV device's IP address, and the Fire TV device will be accessible from your IDEs and other development tools.

Note that this `adb` setup will persist until `adb` is next restarted, even if the Fire TV device is powered down. You can use `adb disconnect` (followed by the IP address) to break the connection if you no longer need it.

Working with the Remote and Controller

Your Fire TV users will be interacting with their Fire TV using the supplied remote, the Fire TV gaming controller, or other controllers.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Wireless Remote

Mostly, the wireless remote offers buttons that you should be handling already, assuming you are implementing a ten-foot UI or are otherwise correctly handling focus management and accessibility. BACK and the D-pad buttons will work on the Fire TV much as you would expect.

The Fire TV device remote has a MENU button, which will bring up an old-style Android options menu. Your simple action bar items, particularly those in the overflow, will appear in this menu without issue. Most likely you will want to skip the action bar on your Fire TV apps, though, which means that action views, action providers, and the like will need to be replaced with alternatives.

The remote also offers play/pause, rewind, and fast-forward buttons that map to their corresponding Android `KeyEvent` types (e.g., `KEYCODE_MEDIA_PLAY_PAUSE`). You can watch for these events in `onKeyDown()` of your activity to be able to respond to them.

Note that you do not have the ability to intercept home or voice search button presses on the wireless remote. And, the Fire TV Stick's remote does not offer the voice search button.

Gaming Controller

The wireless remote's input options are purely buttons. The Fire TV series gaming controller, like most such controllers, are designed for more "analog" input, where the force you apply to a stick might trigger slightly different behavior in a game.

As such, the gaming controller support in Android and FireOS takes a two-tier approach, with primary and secondary ways of handling events. If your code consumes the primary event, the secondary event is not triggered. The idea is that fairly "vanilla" apps might pay attention to the secondary events, as they tend to be more in common with the events you might get from the wireless remote or non-Fire TV devices. But apps that are more game-centric might pay attention to the primary events instead.

For analog inputs — the left and right sticks and the D-pad — the primary event input is supplied as `MotionEvent` objects, which you can pick up in methods like `onGenericMotionEvent()` in a `View`. Secondary event input comes in the form of standard D-pad events. So, an ordinary app will automatically support the sticks and D-pad, simply by accepting D-pad input.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Some buttons also take the two-tier approach. The A button (bottom one in the button cluster on the upper-right side of the controller) can be picked up either as a `KEYCODE_BUTTON_A` `KeyEvent` (primary) or a `KEYCODE_DPAD_CENTER` `KeyEvent` (secondary). The B button (right one) maps to `KEYCODE_BUTTON_B` (primary) and `KEYCODE_BACK` (secondary).

Other buttons just fire simple `KeyEvents` (e.g., left shoulder is `KEYCODE_BUTTON_L1`), while the two triggers just use `MotionEvents` (`AXIS_BRAKE` on the left, `AXIS_GAS` on the right).

Amazon's Fire TV site has full details of the options and how to use them, plus a dedicated game controller API to help you manage multiple controllers and the like.

How Does Distribution Work?

Like the Kindle Fire, Fire TV devices lack the Play Store. If you want your app to be available to Fire TV device users, you will need to explore other ways of promoting and delivering the app.

The principal — and nearly exclusive — way to get apps onto a Fire TV device is by listing them in the Amazon AppStore for Android, much as you would do for Kindle Fire devices.

However, for ordinary users, that is the *only* option. Most Android devices — including the Kindle Fire series — allow the user to download apps from Web sites, if they have the appropriate option checked in Settings. The Fire TV lacks this setting, and therefore ordinary users cannot download an app to the Fire TV from third-party app stores.

"Sideloading" an app using `adb` works, though this is usually only viable for developers or serious power users. Such apps will not appear in the home screen launcher and must be launched instead via Settings > Applications.

Getting Help

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Trail: Appendices

Special Creative Commons BY-NC-SA 4.0 License Edition

CWAC Libraries

CommonsWare — the publisher of this book — has also published a series of open source libraries, collectively named the CommonsWare Android Components (CWAC). If you have read through the book, you will have seen many of these libraries.

This appendix lists all of the CWAC libraries. If the library is covered elsewhere in the book, the appendix links you to that coverage. Those that are not covered elsewhere will be described in this appendix, to accompany the online documentation found at the library's GitHub repository.

cwac-adapter

The cwac-adapter repository contains a small `AdapterWrapper` class that wraps a `ListAdapter`. The default implementation of all `AdapterWrapper` methods is to forward the request along to the wrapped `ListAdapter`. However, you can subclass `AdapterWrapper` to override that behavior.

cwac-camera

The cwac-camera repository holds the CommonsWare library to assist with the use of the hardware cameras on Android devices. This library is covered extensively in the chapter on using the camera.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

cwac-colormixer

The cwac-colormixer repository holds a custom `ColorMixer View`, along with wrappers for using that `View` as a dialog, activity, or preference for use with `PreferenceScreen`.

`ColorMixerDialog` and `ColorPreference` are covered in the chapter on custom dialogs. The `ColorMixer` widget is similar to the implementation found in the chapter on custom views.

This library also contains a `ColorMixerActivity`, which you can use via `startActivityForResult()` to obtain a color, rather than by integrating the widget, dialog, or preference.

cwac-layouts

The cwac-layouts repository contains a series of custom containers and related views.

The current contents of this library — `AspectLockedFrameLayout`, `MirroringFrameLayout`, and kin — are covered in the chapter on custom views.

cwac-mediarouter

The cwac-mediarouter repository contains a cross-port of the `mediarouter-v7` portion of the Android Support package, to eliminate its dependency on `appcompat-v7` and support the native action bar.

cwac-merge

The cwac-merge repository contains `MergeAdapter`. It simply stitches together lots of smaller `ListAdapters` into one larger `ListAdapter` to put inside of a `ListView` or similar `AdapterView`. It also allows you to blend individual "row" `Views` with other `ListAdapters`.

One use of this is for section headers, using row `Views` for the headers and `ListAdapters` for the sections.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

Another use is where you have multiple disparate data sources (e.g., queries across a few databases or `ContentProviders`), each with distinct row formatting, but you want to present them as one contiguous list.

cwac-pager

The cwac-pager repository includes code written in support of the `ViewPager` widget.

`ArrayPagerAdapter` is covered in the chapter on advanced uses of `ViewPager`.

cwac-presentation

The cwac-presentation repository contains code in support of the `Presentation` system, for sending alternative content to an external display, independent of the device's primary screen.

All of the classes in this repository are covered in the chapter on the `Presentation` system.

cwac-provider

The cwac-provider repository contains `StreamProvider`, a riff on Google's `FileProvider`, offering a "canned" implementation of a `ContentProvider` that can serve files from a variety of sources, such as assets and raw resources from your project.

This is discussed briefly in the chapter on `ContentProvider` implementations.

cwac-richedit

The cwac-richedit repository contains the `RichEditText` widget, a drop-in replacement for `EditText` that supports "rich text" (a.k.a., formatted text) editing, such as bold and italics. The use of this widget is covered in the chapter on Android's rich text handling.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition

cwac-sacklist

The cwac-sacklist repository contains `SackOfViewsAdapter`, which implements the `ListAdapter` interface for a collection of individual `Views` that serve as rows. This is used in support of the `MergeAdapter`, for example.

cwac-security

The cwac-security repository contains code to help app developers help their users defend against attacks. At the moment, this contains the `PermissionUtils` class, used to help determine if a custom permission was defined by another app before yours was installed. This is discussed in the chapter on advanced permission techniques.

cwac-strictmodeex

The cwac-strictmodeex repository contains classes that serve a similar role to Android's `StrictMode`, yelling at you for problematic code.

Specifically, this repository contains `StrictAdapter`, which measures the timing of methods like `getView()`, logging information about slow adapters that may result in sluggish `ListView` scrolling.

cwac-wakeful

The cwac-wakeful repository contains the `WakefulIntentService` and related support classes, for doing work and keeping the device awake while that work is going on. This is covered in the chapter on `AlarmManager`.

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition