The Busy Coder’s Guide to Android Development

Mark L. Murphy
# Table of Contents

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

- **Preface**
  - Welcome to the Book! ................................................................. xxxiii
  - The Book’s Structure ................................................................. xxxiii
  - *The Trails* .................................................................................. xxxiv
  - About the Updates ........................................................................ xxxix
  - Warescription ............................................................................... xxxix
  - Getting Help ................................................................................. xl
  - Book Bug Bounty .......................................................................... xl
  - Source Code And Its License ....................................................... xli
  - *Creative Commons and the Four-to-Free (42F) Guarantee* .... xlii
  - Acknowledgments ...................................................................... xlii

- **Key Android Concepts**
  - Android Applications ................................................................. 1
  - Android Devices ........................................................................... 7
  - Don’t Be Scared ........................................................................... 10

- **Choosing Your IDE**
  - Eclipse .......................................................................................... 11
  - Android Studio ............................................................................... 12
  - Alternative IDEs ........................................................................... 12
  - IDEs... And This Book ................................................................. 13
  - About App Inventor ....................................................................... 13

- **Tutorial #1 - Installing the Tools**
  - Step #1 - Checking Your Hardware Requirements ....................... 15
  - Step #2 - Setting Up Java ............................................................. 16
  - Step #3 - Install the Android SDK ............................................... 16
  - Step #4 - Install the ADT for Eclipse .......................................... 18
  - Step #5 - Install Apache Ant ......................................................... 20
  - Step #6 - Set Up the Emulator ..................................................... 21
  - Step #7 - Set Up the Device ......................................................... 28
  - In Our Next Episode... ................................................................. 31

- **Tutorial #2 - Creating a Stub Project**
  - About Our Tutorial Project ......................................................... 33
  - About the Rest of the Tutorials .................................................. 34
  - About the Eclipse Instructions ................................................... 34
  - Step #2: Running the Project ...................................................... 51
<table>
<thead>
<tr>
<th>Contents of Android Projects</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Root Contents</td>
<td>55</td>
</tr>
<tr>
<td>The Sweat Off Your Brow</td>
<td>56</td>
</tr>
<tr>
<td>Resources</td>
<td>56</td>
</tr>
<tr>
<td>What You Get Out Of It</td>
<td>57</td>
</tr>
<tr>
<td>Inside the Manifest</td>
<td></td>
</tr>
<tr>
<td>An Application For Your Application</td>
<td>61</td>
</tr>
<tr>
<td>Specifying Versions</td>
<td>62</td>
</tr>
<tr>
<td>Supporting Multiple Screens</td>
<td>62</td>
</tr>
<tr>
<td>Other Stuff</td>
<td>63</td>
</tr>
<tr>
<td>Tutorial #3 - Changing Our Manifest</td>
<td></td>
</tr>
<tr>
<td>Step #1: Updating the Package Name</td>
<td>65</td>
</tr>
<tr>
<td>Step #2: Supporting Screens</td>
<td>67</td>
</tr>
<tr>
<td>Step #3: Validating our Minimum and Target SDK Versions</td>
<td>70</td>
</tr>
<tr>
<td>In Our Next Episode...</td>
<td>72</td>
</tr>
<tr>
<td>Some Words About Resources</td>
<td></td>
</tr>
<tr>
<td>String Theory</td>
<td>73</td>
</tr>
<tr>
<td>Got the Picture?</td>
<td>77</td>
</tr>
<tr>
<td>Dimensions</td>
<td>81</td>
</tr>
<tr>
<td>The Resource That Shall Not Be Named... Yet</td>
<td>83</td>
</tr>
<tr>
<td>Tutorial #4 - Adjusting Our Resources</td>
<td></td>
</tr>
<tr>
<td>Step #1: Changing the Name</td>
<td>85</td>
</tr>
<tr>
<td>Step #2: Changing the Icon</td>
<td>87</td>
</tr>
<tr>
<td>Step #3: Running the Result</td>
<td>95</td>
</tr>
<tr>
<td>In Our Next Episode...</td>
<td>96</td>
</tr>
<tr>
<td>The Theory of Widgets</td>
<td></td>
</tr>
<tr>
<td>What Are Widgets?</td>
<td>97</td>
</tr>
<tr>
<td>Size, Margins, and Padding</td>
<td>99</td>
</tr>
<tr>
<td>What Are Containers?</td>
<td>99</td>
</tr>
<tr>
<td>The Absolute Positioning Anti-Pattern</td>
<td>100</td>
</tr>
<tr>
<td>The Android User Interface</td>
<td></td>
</tr>
<tr>
<td>The Activity</td>
<td>103</td>
</tr>
<tr>
<td>Dissecting the Activity</td>
<td>104</td>
</tr>
<tr>
<td>Using XML-Based Layouts</td>
<td>105</td>
</tr>
<tr>
<td>Basic Widgets</td>
<td></td>
</tr>
<tr>
<td>Common Concepts</td>
<td>111</td>
</tr>
<tr>
<td>Assigning Labels</td>
<td>113</td>
</tr>
<tr>
<td>A Commanding Button</td>
<td>118</td>
</tr>
<tr>
<td>Fleeting Images</td>
<td>121</td>
</tr>
<tr>
<td>Fields of Green. Or Other Colors.</td>
<td>126</td>
</tr>
</tbody>
</table>
• More Common Concepts ................................................................. 129
• Visit the Trails! ........................................................................... 131

• Debugging Crashes
  © Get Thee To a Stack Trace ....................................................... 134
  © The Case of the Confounding Class Cast ................................. 137
  © Point Break ........................................................................... 137

• Layout and the Box Model
  © Concepts and Properties .......................................................... 139
  © Eclipse Graphical Layout Editor ............................................... 143

• Other Common Widgets and Containers
  © Just a Box to Check .................................................................. 145
  © Don’t Like Checkboxes? How About Toggles? ......................... 148
  © Turn the Radio Up .................................................................. 150
  © All Things Are Relative ........................................................... 152
  © Tabula Rasa ........................................................................... 159
  © Scrollwork ............................................................................. 163
  © Making Progress with ProgressBars ......................................... 166
  © Visit the Trails! ....................................................................... 167

• Tutorial #5 - Making Progress
  © Step #1: Renaming the Activity Class ....................................... 169
  © Step #2: Removing The “Hello, World” ................................... 170
  © Step #3: Adding a ProgressBar ............................................... 171
  © Step #4: Seeing the Results ..................................................... 173
  © In Our Next Episode... ............................................................. 174

• GUI Building, Continued
  © Making Your Selection ........................................................... 175
  © Including Includes ................................................................. 175
  © Wrap It Up (In a Container) ..................................................... 177
  © Morphing Widgets ................................................................. 177
  © Preview of Coming Attractions ............................................... 178

• AdapterViews and Adapters
  © Adapting to the Circumstances ................................................. 179
  © Lists of Naughty and Nice ....................................................... 181
  © Clicks versus Selections .......................................................... 183
  © Spin Control ........................................................................... 187
  © Grid Your Lions (Or Something Like That...) .......................... 190
  © Fields: Now With 35% Less Typing! ........................................ 193
  © Customizing the Adapter ....................................................... 198
  © Visit the Trails! ..................................................................... 206

• The WebView Widget
  © Role of WebView .................................................................... 207
- WebView and WebKit ................................................................. 208
- Adding the Widget ................................................................... 208
- Loading Content Via a URL .................................................... 209
- Supporting JavaScript ............................................................. 211
- Alternatives for Loading Content ............................................. 212
- Listening for Events ............................................................... 213
- WebView and Android 4.4 ....................................................... 216
- Visit the Trails! ....................................................................... 217

**Defining and Using Styles**
- Styles: DIY DRY ..................................................................... 219
- Elements of Style ................................................................... 221
- Themes: Would a Style By Any Other Name... ......................... 224

**JARs and Library Projects**
- The Dalvik VM ...................................................................... 226
- The Easy Part ......................................................................... 226
- The Outer Limits .................................................................... 227
- OK, So What is a Library Project? .......................................... 228
- Creating a Library Project ..................................................... 228
- Using a Library Project .......................................................... 229
- Limitations of Library Projects .............................................. 230
- The Android Support Package ............................................... 231
- JAR Dependency Management .............................................. 233

**Tutorial #6 - Adding a Library**
- Step #1: Downloading and Unpacking ActionBarSherlock .... 235
- Step #2: Adding the Library to Your Project .......................... 236
- In Our Next Episode... ............................................................ 240

**Options Menus and the Action Bar**
- Bar Hopping (a.k.a., Terminology) ....................................... 241
- Yet Another History Lesson ................................................... 246
- Your Action Bar Options ....................................................... 247
- Setting the Target ................................................................ 251
- Minding Narrow ................................................................. 252
- Defining the Resource ......................................................... 252
- Applying the Resource ......................................................... 256
- Responding to Events ......................................................... 256
- Attaching to Action Layouts ................................................... 257
- The Rest of the Sample Activity ........................................... 258
- Floating Action Bars ............................................................ 265
- MENU Key, We Hardly Knew Ye .......................................... 268
- Visit the Trails! .................................................................... 269

**Tutorial #7 - Adding the Action Bar**
- Step #1: Setting the Theme and Splitting the Bar ......................... 271
- Step #2: Changing to SherlockFragmentActivity .......................... 273
- **Step #3: Defining Some Options** ........................................... 275
- Step #4: Loading and Responding to Our Options ........................ 278
- Step #5: Running the Result ..................................................... 279
- In Our Next Episode... ............................................................. 282

- **Android’s Process Model**
  - When Processes Are Created .................................................. 283
  - BACK, HOME, and Your Process ............................................ 284
  - Termination ............................................................................ 285
  - Foreground Means “I Love You” ............................................. 285
  - You and Your Heap ............................................................... 286

- **Activities and Their Lifecycles**
  - Creating Your Second (and Third and...) Activity ...................... 288
  - Warning! Contains Explicit Intents! ......................................... 293
  - Using Implicit Intents ............................................................. 295
  - Extra! Extra! ........................................................................... 300
  - Asynchronicity and Results ..................................................... 302
  - Schroedinger’s Activity .......................................................... 302
  - Life, Death, and Your Activity ................................................. 303
  - When Activities Die .............................................................. 305
  - Walking Through the Lifecycle ............................................... 306
  - Recycling Activities ............................................................. 309

- **Tutorial #8 – Setting Up An Activity**
  - Step #1: Creating the Stub Activity Class ................................. 311
  - Step #2: Adding the Activity to the Manifest ......................... 313
  - Step #3: Launching Our Activity ............................................ 315
  - In Our Next Episode... ............................................................ 316

- **The Tactics of Fragments**
  - The Six Questions .................................................................. 317
  - Your First Fragment .................................................................. 319
  - The Fragment Lifecycle Methods ............................................. 324
  - Your First Dynamic Fragment .................................................. 325
  - Fragments and the Action Bar ................................................ 329
  - Fragments Within Fragments: Just Say “Maybe” ....................... 330
  - Fragments and Multiple Activities ......................................... 331

- **Tutorial #9 - Starting Our Fragments**
  - Step #1: Copy In WebViewFragment ....................................... 333
  - Step #2: Examining WebViewFragment .................................. 337
  - Step #3: Creating AbstractContentViewFragment .................. 337
  - Step #4: Examining AbstractContentViewFragment .................. 339
• **In Our Next Episode** .......................................................... 339
  - **Swiping with ViewPager**
    - Swiping Design Patterns ............................................. 341
    - Paging Fragments ..................................................... 342
    - Paging Other Stuff .................................................. 346
    - Indicators .................................................................... 347
    - Hosting ViewPager in a Fragment ............................... 350
    - Pages and the Action Bar .......................................... 351
    - ViewPagers and Scrollable Contents .......................... 354
  - **Tutorial #10 - Rigging Up a ViewPager**
    - Step #1: Add a ViewPager to the Layout .................... 357
    - Step #2: Obtaining Our ViewPager .............................. 358
    - Step #3: Creating a ContentsAdapter ......................... 359
    - Step #4: Setting Up the ViewPager ............................. 360
    - In Our Next Episode .................................................. 361
  - **Resource Sets and Configurations**
    - What’s a Configuration? And How Do They Change? ... 363
    - Configurations and Resource Sets ............................... 364
    - Screen Size and Orientation ........................................ 365
    - Coping with Complexity ............................................. 368
    - Choosing The Right Resource ..................................... 369
    - Default Change Behavior ......................................... 373
    - Your Options for Configuration Changes ................... 375
    - Blocking Rotations .................................................. 388
  - **Dealing with Threads**
    - The Main Application Thread .................................... 389
    - Getting to the Background ......................................... 390
    - Asyncing Feeling ........................................................ 391
    - Alternatives to AsyncTask .......................................... 400
    - And Now, The Caveats .............................................. 402
  - **Requesting Permissions**
    - Mother, May I? .......................................................... 404
    - New Permissions in Old Applications ........................... 405
    - Permissions: Up Front Or Not At All ........................... 406
    - Signature Permissions ............................................... 407
    - Requiring Permissions .............................................. 407
  - **Assets, Files, and Data Parsing**
    - Packaging Files with Your App ................................... 409
    - Files and Android ..................................................... 411
    - Working with Internal Storage ................................... 412
    - **Working with External Storage** ................................. 415

---

**Subscribe to updates at** https://commonsware.com  
**Special Creative Commons BY-NC-SA 4.0 License Edition**
• Multiple User Accounts .......................................................... 420
• Linux Filesystems: You Sync, You Win ................................... 420
• StrictMode: Avoiding Janky Code .......................................... 422
• XML Parsing Options .............................................................. 428
• JSON Parsing Options ........................................................... 429
• Visit the Trails! ..................................................................... 430

• Tutorial #11 - Adding Simple Content
  ○ Step #1: Adding Some Content ............................................ 431
  ○ Step #2: Create a SimpleContentFragment .......................... 432
  ○ Step #3: Examining SimpleContentFragment ....................... 433
  ○ Step #4: Using SimpleContentFragment ............................. 433
  ○ Step #5: Launching Our Activities, For Real This Time .......... 434
  ○ In Our Next Episode... ...................................................... 436

• Tutorial #12 - Displaying the Book
  ○ Step #1: Adding a Book ...................................................... 437
  ○ Step #2: Defining Our Model .............................................. 438
  ○ Step #3: Examining Our Model .......................................... 440
  ○ Step #4: Creating a ModelFragment ................................... 440
  ○ Step #5: Examining the ModelFragment ............................. 443
  ○ Step #6: Supplying the Content ....................................... 444
  ○ Step #7: Adapting the Content ......................................... 445
  ○ Step #8: Going Home, Again ............................................ 447
  ○ In Our Next Episode... ...................................................... 448

• Using Preferences
  ○ Getting What You Want .................................................... 449
  ○ Stating Your Preference .................................................. 450
  ○ Introducing PreferenceActivity ....................................... 451
  ○ Types of Preferences ...................................................... 463
  ○ Intents for Headers or Preferences ................................... 466
  ○ Conditional Headers ...................................................... 467

• Tutorial #13 - Using Some Preferences
  ○ Step #1: Adding a StockPreferenceFragment ...................... 476
  ○ Step #2: Defining the Preference XML Files ....................... 477
  ○ Step #3: Creating Our PreferenceActivity ........................ 479
  ○ Step #4: Adding To Our Action Bar ................................... 480
  ○ Step #5: Launching the PreferenceActivity ........................ 482
  ○ Step #6: Loading Our Preferences ................................... 486
  ○ Step #7: Saving the Last-Read Position .............................. 488
  ○ Step #8: Restoring the Last-Read Position ........................ 489
  ○ Step #9: Keeping the Screen On ....................................... 489
  ○ In Our Next Episode... ...................................................... 490
• **SQLite Databases**
  ◦ Introducing SQLite ................................................................. 491
  ◦ Thinking About Schemas ....................................................... 492
  ◦ Start with a Helper ............................................................... 492
  ◦ Getting Data Out ................................................................. 497
  ◦ The Rest of the CRUD .......................................................... 503
  ◦ Leveraging ROWID .............................................................. 508
  ◦ Hey, What About Hibernate? ............................................... 509
  ◦ Visit the Trails! ................................................................. 509

• **Tutorial #14 - Saving Notes**
  ◦ Step #1: Adding a DatabaseHelper ........................................ 511
  ◦ Step #2: Examining DatabaseHelper ........................................ 513
  ◦ Step #3: Creating a NoteFragment .......................................... 514
  ◦ Step #4: Examining NoteFragment .......................................... 516
  ◦ Step #5: Creating the NoteActivity ......................................... 516
  ◦ Step #6: Loading and Saving Notes ......................................... 518
  ◦ Step #7: Add Notes to the Action Bar ...................................... 521
  ◦ Step #8: Support Deleting Notes ............................................ 523
  ◦ In Our Next Episode... ....................................................... 531

• **Internet Access**
  ◦ DIY HTTP ............................................................................. 533
  ◦ HTTP via DownloadManager .................................................. 544
  ◦ Using Third-Party JARs ........................................................... 545
  ◦ SSL ...................................................................................... 545
  ◦ Using HTTP Client Libraries .................................................. 546
  ◦ Visit the Trails ........................................................................ 566

• **Intents, Intent Filters, Broadcasts, and Broadcast Receivers**
  ◦ What’s Your Intent? .............................................................. 567
  ◦ Stating Your Intent(ions) ......................................................... 569
  ◦ Responding to Implicit Intents ................................................ 570
  ◦ Requesting Implicit Intents ...................................................... 572
  ◦ Broadcasts and Receivers ....................................................... 576
  ◦ Example System Broadcasts ................................................... 578
  ◦ Downloading Files ............................................................... 585
  ◦ The Order of Things ............................................................. 597
  ◦ Keeping It Local ..................................................................... 598

• **Tutorial #15 - Sharing Your Notes**
  ◦ Step #1: Adding a Share Action Bar Item ................................. 599
  ◦ Step #2: Sharing the Note ....................................................... 600
  ◦ Step #3: Tying Them Together ................................................ 601
  ◦ Step #4: Testing the Result ..................................................... 601
• In Our Next Episode................................................................. 602

• Services and the Command Pattern
  ◦ Why Services? ..................................................................... 603
  ◦ Setting Up a Service ............................................................. 604
  ◦ Communicating To Services ............................................... 606
  ◦ Scenario: The Music Player ................................................ 608
  ◦ Communicating From Services .......................................... 611
  ◦ Scenario: The Downloader ................................................. 613

• Tutorial #16 - Updating the Book
  ◦ Step #1: Adding a Stub DownloadCheckService ............... 620
  ◦ Step #2: Tying the Service Into the Action Bar ................. 621
  ◦ Step #3: Adding a Stub DownloadCompleteReceiver .......... 622
  ◦ Step #4: Completing the DownloadCheckService ............. 623
  ◦ Step #5: Adding a Stub DownloadInstallService ............... 627
  ◦ Step #6: Completing the DownloadCompleteReceiver .......... 628
  ◦ Step #7: Completing the DownloadInstallService ............. 629
  ◦ Step #8: Updating ModelFragment .................................... 631
  ◦ Step #9: Adding a BroadcastReceiver to EmPubLiteActivity .... 634
  ◦ Step #10: Discussing the Flaws ......................................... 638
  ◦ In Our Next Episode........................................................... 638

• AlarmManager and the Scheduled Service Pattern
  ◦ Scenarios ............................................................................. 639
  ◦ Options ................................................................................. 640
  ◦ A Simple Example ............................................................... 642
  ◦ The Five set...() Varieties .................................................. 644
  ◦ The Four Types of Alarms ................................................... 646
  ◦ When to Schedule Alarms .................................................... 646
  ◦ Get Moving, First Thing ..................................................... 648
  ◦ Archetype: Scheduled Service Polling ............................. 651
  ◦ Staying Awake at Work ....................................................... 655
  ◦ Warning: Not All Android Devices Play Nice ..................... 658
  ◦ Debugging Alarms .............................................................. 659
  ◦ WakefulBroadcastReceiver .............................................. 666

• Tutorial #17 - Periodic Book Updates
  ◦ Step #1: Adding a Stub UpdateReceiver ............................ 671
  ◦ Step #2: Scheduling the Alarms ......................................... 673
  ◦ Step #3: Adding the WakefulIntentService ....................... 674
  ◦ Step #4: Using WakefulIntentService ............................... 675
  ◦ Step #5: Completing the UpdateReceiver .......................... 676
  ◦ In Our Next Episode........................................................... 676

• Notifications

ix
• What’s a Notification? ........................................................... 677
• Showing a Simple Notification ............................................. 679
• Seeking Some Order ............................................................. 684
• Big (and Rich) Notifications .................................................. 690
• Foreground Services .............................................................. 697
• Disabled Notifications ........................................................... 701

• Tutorial #18 - Notifying the User
  ▪ Step #1: Adding the InstallReceiver ...................................... 703
  ▪ Step #2: Completing the InstallReceiver ................................. 705
  ▪ In Our Next Episode... ........................................................... 706

• Large-Screen Strategies and Tactics
  ▪ Objective: Maximum Gain, Minimum Pain ............................. 707
  ▪ The Fragment Strategy .......................................................... 707
  ▪ Fragment Example: The List-and-Detail Pattern ....................... 716
  ▪ Other European Flavors ........................................................ 728
  ▪ Showing More Pages ............................................................ 741
  ▪ Fragment FAQs .................................................................... 745
  ▪ Screen Size and Density Tactics ............................................. 746
  ▪ Other Considerations ............................................................. 749

• Tutorial #19 - Supporting Large Screens
  ▪ Step #1: Creating Our Layouts ............................................... 753
  ▪ Step #2: Loading Our Sidebar Widgets .................................. 757
  ▪ Step #3: Opening the Sidebar ............................................... 758
  ▪ Step #4: Loading Content Into the Sidebar .............................. 759
  ▪ Step #5: Removing Content From the Sidebar ......................... 762

• Backwards Compatibility Strategies and Tactics
  ▪ Think Forwards, Not Backwards ............................................. 765
  ▪ Aim Where You Are Going .................................................... 767
  ▪ A Target-Rich Environment ................................................... 767
  ▪ Lint: It’s Not Just For Belly Buttons ........................................ 769
  ▪ A Little Help From Your Friends ............................................ 769
  ▪ Avoid the New on the Old ....................................................... 770
  ▪ Testing .................................................................................. 774
  ▪ Keeping Track of Changes ..................................................... 774

• Getting Help
  ▪ Questions. Sometimes, With Answers. .................................... 777
  ▪ Heading to the Source .......................................................... 778
  ▪ Getting Your News Fix ......................................................... 779

• Introducing GridLayout
  ▪ Prerequisites .................................................................... 781
  ▪ Issues with the Classic Containers ........................................ 781
• The New Contender: GridLayout ................................................................. 783
• GridLayout and the Android Support Package ........................................ 784
• Eclipse and GridLayout ........................................................................... 785
• Trying to Have Some Rhythm ............................................................... 785
• Our Test App ......................................................................................... 786
• Replacing the Classics ........................................................................... 788
• Implicit Rows and Columns .................................................................... 795
• Row and Column Spans ......................................................................... 797
• Should You Use GridLayout? ............................................................... 801

• Dialogs and DialogFragments
  • Prerequisites ....................................................................................... 803
  • DatePickerDialog and TimePickerDialog ............................................. 803
  • AlertDialog ......................................................................................... 809
  • DialogFragments ................................................................................ 810
  • DialogFragment: The Other Flavor .................................................... 814
  • Dialogs: Modal, Not Blocking ........................................................... 815

• Advanced ListViews
  • Prerequisites ....................................................................................... 817
  • Multiple Row Types, and Self Inflation ............................................... 817
  • Choice Modes and the Activated Style ............................................... 823
  • Custom Mutable Row Contents .......................................................... 824
  • From Head To Toe ............................................................................... 830

• Action Bar Navigation
  • Prerequisites ....................................................................................... 835
  • List Navigation .................................................................................... 835
  • Tabs (And Sometimes List) Navigation ............................................... 840
  • Custom Navigation ............................................................................... 846

• Action Modes and Context Menus
  • Prerequisites ....................................................................................... 848
  • Another Wee Spot O’ History .............................................................. 848
  • Manual Action Modes ......................................................................... 849
  • Multiple-Modal-Choice Action Modes ............................................... 854
  • Long-Click To Initiate an Action Mode ............................................... 859
  • Split Action Modes ............................................................................... 864
  • What Came Before: Context Menus .................................................. 866

• ActionBarCompat: The Official Action Bar Backport
  • Prerequisites ....................................................................................... 869
  • Using the ActionBarCompat ............................................................. 869
  • Choosing a Backport ............................................................................ 875

• Other Advanced Action Bar Techniques
  • Prerequisites ....................................................................................... 877
• Action Layouts, Action Views, and Action Providers ............... 877
• Searching with SearchView .................................................. 878

  Implementing a Navigation Drawer
  ○ Prerequisites ................................................................. 885
  ○ What is a Navigation Drawer? ............................................. 885
  ○ A Simple Navigation Drawer ............................................. 887
  ○ Alternative Row Layouts .................................................. 893
  ○ Additional Considerations .............................................. 895
  ○ What Should Not Be in the Drawer ................................... 904
  ○ Independent Implementations .......................................... 904

  Advanced Uses of WebView
  ○ Prerequisites ................................................................. 907
  ○ Friends with Benefits ..................................................... 907
  ○ Turnabout is Fair Play ..................................................... 913
  ○ Navigating the Waters ..................................................... 917
  ○ Settings, Preferences, and Options (Oh, My!) ...................... 918

  The Input Method Framework
  ○ Prerequisites ................................................................. 919
  ○ Keyboards, Hard and Soft ................................................. 919
  ○ Tailored To Your Needs ................................................... 920
  ○ Tell Android Where It Can Go ......................................... 924
  ○ Fitting In ........................................................................ 926
  ○ Jane, Stop This Crazy Thing! ............................................. 928

  Fonts
  ○ Prerequisites ................................................................. 931
  ○ Love The One You’re With ............................................... 931
  ○ Here a Glyph, There a Glyph ............................................ 935

  Rich Text
  ○ Prerequisites ................................................................. 937
  ○ The Span Concept .......................................................... 937
  ○ Loading Rich Text .......................................................... 939
  ○ Editing Rich Text ............................................................ 941
  ○ Saving Rich Text .............................................................. 946
  ○ Manipulating Rich Text .................................................... 947

  Custom Drawables
  ○ Prerequisites ................................................................. 949
  ○ ColorDrawable ............................................................... 950
  ○ AnimationDrawable ........................................................ 950
  ○ StateListDrawable .......................................................... 953
  ○ LayerDrawable ............................................................... 955
  ○ TransitionDrawable ........................................................ 956
• LevelListDrawable ................................................................. 957
• ScaleDrawable and ClipDrawable ........................................ 958
• InsetDrawable ................................................................. 967
• ShapeDrawable ................................................................. 968
• BitmapDrawable ................................................................. 978
• Composite Drawables ...................................................... 985
• XML Drawables and Eclipse .............................................. 989
• A Stitch In Time Saves Nine .............................................. 989

• Animators
  ◦ Prerequisites ........................................................................ 999
  ◦ ViewPropertyAnimator ..................................................... 999
  ◦ The Foundation: Value and Object Animators ..................... 1004
  ◦ Animating Custom Types .................................................. 1007
  ◦ Hardware Acceleration ..................................................... 1008
  ◦ The Three-Fragment Problem ........................................... 1009

• Legacy Animations
  ◦ Prerequisites ........................................................................ 1021
  ◦ It’s Not Just For Toons Anymore ....................................... 1021
  ◦ A Quirky Translation ......................................................... 1022
  ◦ Fading To Black. Or Some Other Color. ........................... 1026
  ◦ When It’s All Said And Done ........................................... 1028
  ◦ Loose Fill ........................................................................... 1029
  ◦ Hit The Accelerator .......................................................... 1029
  ◦ Animate. Set. Match. ........................................................ 1030
  ◦ Active Animations ............................................................. 1031

• Mapping with Maps V2
  ◦ Prerequisites ........................................................................ 1033
  ◦ A Brief History of Mapping on Android ............................. 1034
  ◦ Where You Can Use Maps V2 ........................................... 1035
  ◦ Licensing Terms for Maps V2 ............................................ 1035
  ◦ What You Need to Start .................................................. 1036
  ◦ The Book Samples... And You! ........................................ 1039
  ◦ Setting Up a Basic Map ..................................................... 1039
  ◦ Playing with the Map ......................................................... 1046
  ◦ Placing Simple Markers .................................................. 1051
  ◦ Seeing All the Markers ..................................................... 1054
  ◦ Flattening and Rotating Markers ..................................... 1056
  ◦ Sprucing Up Your “Info Windows” ................................. 1060
  ◦ Images and Your Info Window ....................................... 1064
  ◦ Setting the Marker Icon .................................................. 1071
  ◦ Responding to Taps .......................................................... 1072
• Dragging Markers ................................................................. 1074
• The “Final” Limitations ........................................................ 1076
• A Bit More About IPC ............................................................ 1079
• Finding the User ...................................................................... 1080
• Drawing Lines and Areas ....................................................... 1084
• Gestures and Controls ............................................................ 1087
• Tracking Camera Areas ............................................................ 1087
• Maps in Fragments and Pagers ............................................... 1090
• Animating Marker Movement .................................................. 1095
• Maps, of the Indoor Variety ..................................................... 1103
• Taking a Snapshot of a Map ..................................................... 1104
• MapFragment vs. MapView ..................................................... 1105
• Maps and ActionBarSherlock .................................................. 1105
• About That AbstractMapActivity Class ................................... 1108
• Helper Libraries for Maps V2 ................................................. 1112
• Problems with Maps V2 at Runtime ........................................ 1116
• Problems with Maps V2 Deployment ........................................ 1116
• What Non-Compliant Devices Show ....................................... 1116
• Mapping Alternatives ............................................................. 1117
• News and Getting Help ............................................................ 1117

• Crafting Your Own Views
  • Prerequisites ........................................................................ 1119
  • Pick Your Poison ................................................................... 1119
  • Colors, Mixed How You Like Them ........................................ 1121
  • ReverseChronometer: Simply a Custom Subclass .................... 1131
  • AspectLockedFrameLayout: A Custom Container ..................... 1136
  • Mirror and MirroringFrameLayout: Draw It Yourself ............. 1139

• Custom Dialogs and Preferences
  • Prerequisites ........................................................................ 1149
  • Your Dialog, Chocolate-Covered ............................................ 1149
  • Preferring Your Own Preferences, Preferably .......................... 1153

• Progress Indicators
  • Prerequisites ........................................................................ 1161
  • Progress Bars ....................................................................... 1161
  • ProgressBar and Threads ....................................................... 1164
  • Tailoring Progress Bars ........................................................ 1167
  • Progress Dialogs .................................................................. 1175
  • Title Bar and Action Bar Progress Indicators ........................... 1177
  • Action Bar Refresh-and-Progress Items ................................... 1179
  • Direct Progress Indication ....................................................... 1182

• Advanced Notifications
• Prerequisites ............................................................... 1185
• Custom Views: or How Those Progress Bars Work ................ 1185
• Seeing It In Action ................................................... 1187
• How You Really Do Progress Notifications ..................... 1192
• Life After Delete ...................................................... 1195
• The Mysterious Case of the Missing Number .................... 1196
• More Fun with Pagers
  ◦ Prerequisites .................................................................. 1197
  ◦ ViewPager with Action Bar Tabs .................................. 1197
  ◦ Using ViewPagerIndicator .......................................... 1201
  ◦ Columns for Large, Pages for Small .............................. 1205
  ◦ Introducing ArrayPagerAdapter ................................... 1211
  ◦ Columns for Large Landscape, Pages for the Rest ............ 1214
  ◦ Adding, Removing, and Moving Pages ......................... 1219
  ◦ Inside ArrayPagerAdapter ........................................... 1223
• Focus Management and Accessibility
  ◦ Prerequisites .................................................................. 1237
  ◦ Prepping for Testing .................................................. 1238
  ◦ Controlling the Focus ................................................ 1238
  ◦ Accessibility and Focus .............................................. 1247
  ◦ Accessibility Beyond Focus ........................................ 1248
  ◦ Accessibility Beyond Impairment ................................. 1258
• Miscellaneous UI Tricks
  ◦ Prerequisites .................................................................. 1261
  ◦ Full-Screen and Lights-Out Modes ................................. 1261
  ◦ Offering a Delayed Timeout ....................................... 1272
• Event Buses
  ◦ Prerequisites .................................................................. 1277
  ◦ A Brief Note About the Sample Apps ......................... 1277
  ◦ What Is an Event Bus? .............................................. 1278
  ◦ Standard Intents as Event Bus ................................. 1278
  ◦ LocalBroadcastManager as Event Bus ..................... 1279
  ◦ Square’s Otto ....................................................... 1289
  ◦ greenrobot’s EventBus ............................................. 1295
• Home Screen App Widgets
  ◦ Prerequisites .................................................................. 1303
  ◦ East is East, and West is West... .............................. 1304
  ◦ The Big Picture for a Small App Widget .................... 1304
  ◦ Crafting App Widgets .............................................. 1305
  ◦ Another and Another ............................................... 1312
  ◦ App Widgets: Their Life and Times ......................... 1313
• Controlling Your (App Widget's) Destiny ........................................ 1313
• Change Your Look ........................................................................ 1314
• One Size May Not Fit All .............................................................. 1315
• Lockscreen Widgets ....................................................................... 1322
• Preview Images ............................................................................... 1328
• Being a Good Host .......................................................................... 1330

- Adapter-Based App Widgets
  • Prerequisites ............................................................................. 1331
  • AdapterViews for App Widgets .................................................... 1331
  • Building Adapter-Based App Widgets .......................................... 1332

- Content Provider Theory
  • Prerequisites ............................................................................. 1347
  • Using a Content Provider .......................................................... 1347
  • Building Content Providers ......................................................... 1355
  • Issues with Content Providers ................................................... 1361

- Content Provider Implementation Patterns
  • Prerequisites ............................................................................. 1363
  • The Single-Table Database-Backed Content Provider ................. 1363
  • The Local-File Content Provider ................................................ 1371
  • The Protected Provider ............................................................... 1378
  • The Stream Provider .................................................................. 1380
  • FileProvider ............................................................................... 1383
  • StreamProvider ......................................................................... 1387

- The Loader Framework
  • Prerequisites ............................................................................. 1389
  • Cursors: Issues with Management .............................................. 1390
  • Introducing the Loader Framework ............................................. 1390
  • Honeycomb... Or Not ................................................................ 1392
  • Using CursorLoader .................................................................... 1392
  • What Else Is Missing? ................................................................. 1394
  • Issues, Issues, Issues ................................................................. 1395
  • Loaders Beyond Cursors ............................................................ 1395
  • What Happens When...? ............................................................. 1395

- The ContactsContract Provider
  • Prerequisites ............................................................................. 1399
  • Introducing You to Your Contacts .............................................. 1400
  • Pick a Peck of Pickled People ..................................................... 1401
  • Spin Through Your Contacts ..................................................... 1404
  • Makin’ Contacts ........................................................................ 1413

- The CalendarContract Provider
  • Prerequisites ............................................................................. 1420
• You Can’t Be a Faker ................................................................. 1420
• Do You Have Room on Your Calendar? ...................................... 1420
• Penciling In an Event ................................................................. 1425

• The MediaStore Provider
  • Prerequisites ........................................................................... 1427
  • What Is the MediaStore? ............................................................ 1428
  • MediaStore and “Other” External Storage ................................. 1429
  • How Does My Content Get Indexed? ......................................... 1430
  • How Do I Retrieve Video from the MediaStore? ...................... 1430

• Encrypted Storage
  • Prerequisites ........................................................................... 1440
  • Scenarios for Encryption .......................................................... 1440
  • Obtaining SQLCipher ............................................................... 1441
  • Employing SQLCipher ............................................................. 1441
  • SQLCipher Limitations .............................................................. 1444
  • Passwords and Sessions ......................................................... 1445
  • About Those Passphrases... .................................................... 1445
  • Encrypted Preferences .............................................................. 1453
  • IOCipher .................................................................................. 1455

• Packaging and Distributing Data
  • Prerequisites ........................................................................... 1457
  • Packing a Database To Go ......................................................... 1457

• Audio Playback
  • Prerequisites ........................................................................... 1463
  • Get Your Media On .................................................................. 1463
  • MediaPlayer for Audio ............................................................ 1464
  • Other Ways to Make Noise ....................................................... 1470

• Audio Recording
  • Prerequisites ........................................................................... 1473
  • Recording by Intent .................................................................. 1473
  • Recording to Files ..................................................................... 1476
  • Recording to Streams ............................................................... 1479
  • Raw Audio Input ...................................................................... 1482
  • Requesting the Microphone ...................................................... 1482

• Video Playback
  • Prerequisites ........................................................................... 1485
  • Moving Pictures ...................................................................... 1485

• Using the Camera via 3rd-Party Apps
  • Prerequisites ........................................................................... 1491
  • Being Specific About Features ................................................ 1491
  • Still Photos: Letting the Camera App Do It .............................. 1492

xvii
Scanning with ZXing ................................................................. 1494
Videos: Letting the Camera App Do It ........................................ 1495
Directly Working with the Camera .............................................. 1497

• Working Directly with the Camera
  ○ Prerequisites ........................................................................ 1499
  ○ Basic CameraFragment Usage ............................................. 1500
  ○ Simple Configuration and Usage ......................................... 1501
  ○ **Core Camera Concepts** ..................................................... 1508
  ○ Advanced CWAC-Camera Features .................................... 1531

• Media Routes
  ○ Prerequisites ........................................................................ 1539
  ○ Terminology ........................................................................ 1539
  ○ A Tale of Three MediaRouters ............................................ 1540
  ○ Attaching to MediaRouter ................................................... 1542
  ○ User Route Selection with MediaRouteActionProvider ........ 1543
  ○ Using Live Audio Routes .................................................... 1560
  ○ Using Live Video Routes .................................................... 1560
  ○ **Using Remote Playback Routes** ....................................... 1560

• Supporting External Displays
  ○ Prerequisites ........................................................................ 1579
  ○ A History of external displays ............................................. 1579
  ○ What is a Presentation? ......................................................... 1580
  ○ Playing with External Displays .......................................... 1581
  ○ Detecting Displays ............................................................ 1587
  ○ A Simple Presentation ........................................................ 1588
  ○ **A Simpler Presentation** ..................................................... 1593
  ○ Presentations and Configuration Changes .......................... 1598
  ○ Presentations as Fragments ................................................ 1599
  ○ Another Sample Project: Slides .......................................... 1609
  ○ Device Support for Presentation ......................................... 1616
  ○ Hey, What About Chromecast? ........................................... 1617

• **Google Cast and Chromecast**
  ○ Prerequisites ........................................................................ 1619
  ○ Here a Cast, There a Cast ................................................... 1619
  ○ Common Chromecast Development Notes .......................... 1621
  ○ Your API Choices .................................................................. 1621
  ○ Senders and Receivers ........................................................ 1622
  ○ Supported Media Types ....................................................... 1623
  ○ Cast SDK Dependencies ...................................................... 1624
  ○ Developing Google Cast Apps .............................................. 1626

• **The “Ten-Foot UI”**
- **Prerequisites** ................................................................. 1627
- **What is the “Ten-Foot UI”?** ......................................... 1628
- **Overscan** ......................................................................... 1628
- **Navigation** ......................................................................... 1631
- **Stylistic Considerations** ................................................. 1631
- **Testing Your Theories** ..................................................... 1633

- **SSL**
  - Prerequisites ......................................................................... 1635
  - Basic SSL Operation .............................................................. 1635
  - **Common SSL Problems** .................................................. 1636
  - **TrustManagers** ................................................................ 1637
  - **TrustManagerBuilder** ...................................................... 1640
  - **About That Man in the Middle** ....................................... 1643
  - **Self-Signed Certificates, Revisited** .................................. 1645
  - **Certificate Memorizing** .................................................. 1646
  - **Pinning** ........................................................................... 1650
  - **NetCipher** ......................................................................... 1651

- **Advanced Permissions**
  - Prerequisites ......................................................................... 1653
  - Securing Yourself ................................................................. 1653
  - Signature Permissions ........................................................... 1656
  - The Custom Permission Vulnerability ................................... 1658

- **Restricted Profiles and UserManager**
  - Prerequisites ......................................................................... 1669
  - Android Tablets and Multiple User Accounts ......................... 1669
  - Determining What the User Can Do ..................................... 1675
  - Impacts of Device-Level Restrictions .................................... 1678
  - Enabling Custom Restrictions .............................................. 1678
  - Implicit Intents May Go “Boom” .......................................... 1689
  - The Future: App Ops? ............................................................ 1689

- **Tapjacking**
  - Prerequisites ......................................................................... 1691
  - What is Tapjacking? ............................................................... 1691
  - Detecting Potential Tapjackers ............................................. 1696
  - Defending Against Tapjackers .............................................. 1698
  - Why Is This Being Discussed? ............................................. 1701
  - What Changed in 4.0.3? ....................................................... 1702

- **Miscellaneous Security Techniques**
  - Prerequisites ......................................................................... 1703
  - **Public Key Validation** ..................................................... 1703
  - Choosing Your Signing Keysize .......................................... 1719
• Avoiding Accidental APIs ................................................................. 1720
• Other Ways to Expose Data ................................................................. 1725

• Accessing Location-Based Services
  • Prerequisites ...................................................................................... 1729
  • Location Providers: They Know Where You’re Hiding ...................... 1730
  • Finding Yourself ................................................................................. 1730
  • On the Move ...................................................................................... 1732
  • Are We There Yet? Are We There Yet? Are We There Yet? .............. 1733
  • Testing... Testing... ............................................................................ 1734
  • Alternative Flavors of Updates ............................................................ 1735
  • The Fused Option ................................................................................ 1736

• The Fused Location Provider
  • Prerequisites ...................................................................................... 1737
  • Why Use the Fused Location Provider? .............................................. 1737
  • Why Not Use the Fused Location Provider? ........................................ 1738
  • Finding Our Location, Once ................................................................. 1738
  • Requesting Location Updates ............................................................. 1746
  • Gaps in the Fused Location Provider .................................................. 1748

• Working with the Clipboard
  • Prerequisites ...................................................................................... 1749
  • Using the Clipboard on Android 1.x/2.x .............................................. 1749
  • Advanced Clipboard on Android 3.x and Higher .................................. 1752
  • Monitoring the Clipboard ................................................................. 1757
  • The Android 4.3 Clipboard Bug .......................................................... 1759

• Telephony
  • Prerequisites ...................................................................................... 1761
  • Report To The Manager ....................................................................... 1762
  • You Make the Call! .............................................................................. 1762
  • No, Really, You Make the Call! ............................................................ 1765

• Working With SMS
  • Prerequisites ...................................................................................... 1767
  • Sending Out an SOS, Give or Take a Letter ......................................... 1768
  • Monitoring and Receiving SMS ........................................................... 1775
  • The SMS Inbox ................................................................................... 1781
  • Asking to Change the Default .............................................................. 1782
  • SMS and the Emulator ........................................................................ 1783

• NFC
  • Prerequisites ...................................................................................... 1785
  • What Is NFC? ..................................................................................... 1785
  • To NDEF, Or Not to NDEF ................................................................. 1787
  • NDEF Modalities ................................................................................ 1787
• NDEF Structure and Android’s Translation .................................. 1788
• The Reality of NDEF ............................................................. 1789
• Sources of Tags .................................................................... 1791
• Writing to a Tag .................................................................... 1791
• Responding to a Tag ............................................................ 1799
• Expected Pattern: Bootstrap ............................................... 1800
• Mobile Devices are Mobile ................................................ 1801
• Enabled and Disabled ........................................................ 1801
• Android Beam ..................................................................... 1802
• Beaming Files ..................................................................... 1808
• Another Sample: SecretAgentMan ..................................... 1810
• Additional Resources ......................................................... 1819

• **Device Administration**
  • Prerequisites ....................................................................... 1821
  • Objectives and Scope ........................................................ 1821
  • Defining and Registering an Admin Component .............. 1822
  • Going Into Lockdown ........................................................ 1828
  • Passwords and Device Administration ........................ 1835
  • Getting Along with Others ............................................. 1839

• **PowerManager and WakeLocks**
  • Prerequisites ....................................................................... 1841
  • Keeping the Screen On, UI-Style ..................................... 1841
  • The Role of the WakeLock ................................................ 1842
  • What WakefulIntentService Does .................................. 1843

• **Push Notifications with GCM**
  • **Prerequisites** ................................................................. 1845
  • The Precursor: C2DM .......................................................... 1846
  • The Replacement: GCM ...................................................... 1846
  • The Re-Replacement: GCM 2013 ..................................... 1846
  • The Pieces of Push ............................................................ 1847
  • A Simple Push .................................................................... 1853
  • Message Options and Advanced Features ..................... 1867
  • Re-Registration .................................................................. 1868
  • Pre-Release Features ........................................................ 1869
  • Considering Encryption .................................................... 1871
  • Issues with GCM ............................................................... 1872
  • Amazon Simple Notification Service and GCM ........... 1873

• **Basic Use of Sensors**
  • Prerequisites ....................................................................... 1875
  • The Sensor Abstraction Model ......................................... 1875
  • Considering Rates ............................................................. 1876
• Reading Sensors ............................................................... 1877
• Batching Sensor Readings ................................................. 1887

• Other System Settings and Services
  • Prerequisites ......................................................................... 1889
  • Setting Expectations .............................................................. 1889
  • Can You Hear Me Now? OK, How About Now? ......................... 1894
  • The Rest of the Gang ............................................................. 1897

• Dealing with Different Hardware
  • Prerequisites ......................................................................... 1899
  • Filtering Out Devices ............................................................ 1899
  • Runtime Capability Detection ................................................. 1902
  • Dealing with Device Bugs ..................................................... 1903

• Responding to URLs
  • Prerequisites ......................................................................... 1905
  • Manifest Modifications ........................................................ 1905
  • Creating a Custom URL ......................................................... 1907
  • Reacting to the Link ............................................................... 1908

• Plugin Patterns
  • Prerequisites ......................................................................... 1911
  • Definitions, Scenarios, and Scope .......................................... 1911
  • The Keys to Any Plugin System ............................................. 1912
  • Case Study: DashClock .......................................................... 1920
  • Other Plugin Examples .......................................................... 1923

• PackageManager Tricks
  • Prerequisites ......................................................................... 1941
  • Asking Around ...................................................................... 1941
  • Preferred Activities .............................................................. 1945
  • Middle Management ............................................................ 1950

• Searching with SearchManager
  • Prerequisites ......................................................................... 1953
  • Hunting Season .................................................................... 1953
  • Search Yourself .................................................................... 1955
  • Searching for Meaning In Randomness ................................... 1962
  • May I Make a Suggestion? ..................................................... 1963
  • Putting Yourself (Almost) On Par with Google ......................... 1967

• Handling System Events
  • Prerequisites ......................................................................... 1973
  • I Sense a Connection Between Us......................................... 1973
  • Feeling Drained .................................................................... 1975

• Remote Services and the Binding Pattern
  • Prerequisites ......................................................................... 1983
• The Binding Pattern ................................................................. 1984
• When IPC Attacks! ................................................................. 1992
• Service From Afar ................................................................. 1994
• Servicing the Service ............................................................. 1999
• Thinking About Security ....................................................... 2005
• The “Everlasting Service” Anti-Pattern ................................. 2005

• Advanced Manifest Tips
  ◦ Prerequisites ........................................................................ 2007
  ◦ Just Looking For Some Elbow Room ...................................... 2007
  ◦ Using an Alias ...................................................................... 2014
  ◦ Getting Meta (Data) .............................................................. 2016

• Miscellaneous Integration Tips
  ◦ Prerequisites ........................................................................ 2021
  ◦ Take the Shortcut .................................................................. 2021
  ◦ Homing Beacons for Intents .................................................. 2028
  ◦ ShareActionProvider ............................................................ 2028

• Reusable Components
  ◦ Prerequisites ........................................................................ 2033
  ◦ Where Do I Find Them? ......................................................... 2033
  ◦ How Are They Packaged? ...................................................... 2034
  ◦ How Do I Create Them? ......................................................... 2035
  ◦ The Future: AAR ................................................................. 2038
  ◦ Other Considerations for Publishing Reusable Code ............ 2038

• The Role of Scripting Languages
  ◦ Prerequisites ........................................................................ 2043
  ◦ All Grown Up ........................................................................ 2043
  ◦ Following the Script ............................................................... 2044
  ◦ Going Off-Script .................................................................... 2045

• The Scripting Layer for Android
  ◦ Prerequisites ........................................................................ 2049
  ◦ The Role of SL4A ................................................................. 2049
  ◦ Getting Started with SL4A ..................................................... 2050
  ◦ Writing SL4A Scripts ............................................................. 2058
  ◦ Running SL4A Scripts ........................................................... 2063
  ◦ Potential Issues ..................................................................... 2066

• JVM Scripting Languages
  ◦ Prerequisites ........................................................................ 2069
  ◦ Languages on Languages ..................................................... 2069
  ◦ A Brief History of JVM Scripting .......................................... 2070
  ◦ Limitations ........................................................................... 2071
  ◦ SL4A and JVM Languages .................................................. 2072

 xxiii
• Embedding JVM Languages ............................................. 2072
• Other JVM Scripting Languages ..................................... 2086
• JUnit and Android
  ◦ Prerequisites ............................................................. 2089
  ◦ You Get What They Give You ..................................... 2089
  ◦ Your Test Cases ........................................................ 2092
  ◦ Your Test Suite .......................................................... 2097
  ◦ Running Your Tests .................................................... 2098
• MonkeyRunner and the Test Monkey
  ◦ Prerequisites ............................................................. 2101
  ◦ MonkeyRunner .......................................................... 2101
  ◦ Monkeying Around ...................................................... 2103
• Testing with UIAutomator
  ◦ Prerequisites ............................................................. 2105
  ◦ What Is UIAutomator? .................................................. 2105
  ◦ Why Choose UIAutomator Over Alternatives? ............... 2105
  ◦ Creating Some Tests ................................................... 2106
  ◦ Running Your Tests ..................................................... 2115
  ◦ Finding Your Widgets ................................................... 2116
  ◦ Limitations of uiautomator ............................................ 2116
• Introducing Gradle
  ◦ Prerequisites and Warnings .......................................... 2119
  ◦ The Big Questions ....................................................... 2119
  ◦ Obtaining Gradle ........................................................ 2122
  ◦ Versions of Gradle and Gradle for Android ................... 2123
  ◦ Gradle Environment Variables ...................................... 2124
  ◦ Some Brief Words About Maven .................................... 2124
  ◦ Learning More About Gradle ......................................... 2124
• Gradle and Legacy Projects
  ◦ Prerequisites and Warnings .......................................... 2125
  ◦ “Legacy”? ................................................................... 2125
  ◦ Creating Your Gradle Build File ................................... 2126
  ◦ Running a Gradle Build ................................................. 2131
  ◦ Examining the Gradle File ............................................ 2133
• Gradle and the New Project Structure
  ◦ Prerequisites and Warnings .......................................... 2141
  ◦ Objectives of the New Project Structure ......................... 2142
  ◦ Terminology .............................................................. 2142
  ◦ Creating a Project in the New Structure ......................... 2146
  ◦ What the New Project Structure Looks Like ................... 2146
  ◦ Configuring the Stock Build Types ................................ 2149
• Adding Build Types ................................................................. 2155
• Adding Product Flavors and Getting Build Variants ........... 2157
• Revisiting the Legacy Gradle File ........................................ 2161
• **Gradle and Dependencies**
  ◦ Prerequisites and Warnings .................................................. 2163
  ◦ “Dependencies”? ................................................................. 2164
  ◦ The Dependencies Block... and the Other Dependencies Block .. 2164
  ◦ Depending Upon a JAR ....................................................... 2164
  ◦ Depending Upon NDK Binaries ........................................... 2166
  ◦ Depending Upon an Android Library Project ....................... 2166
  ◦ Depending Upon Sub-Projects ............................................. 2167
  ◦ Depending Upon Artifacts ................................................... 2169
  ◦ Creating Android JARs from Gradle .................................... 2176
  ◦ A Property of Transitive (Dependencies) .............................. 2177
  ◦ Dependencies By Build Type ............................................. 2177
  ◦ Dependencies By Flavor .................................................... 2178
  ◦ **Examining Some CWAC Builds** ....................................... 2179
• **Gradle and Testing**
  ◦ Prerequisites and Warnings .................................................. 2185
  ◦ JUnit/Instrumentation Testing ............................................ 2186
  ◦ Other Types of Testing ...................................................... 2191
• **Advanced Gradle for Android Tips**
  ◦ **Prerequisites** .................................................................. 2193
  ◦ Supporting AIDL ................................................................. 2193
  ◦ Supporting the NDK ........................................................... 2195
  ◦ Gradle, DRY ....................................................................... 2201
  ◦ Automating APK Version Information ................................. 2206
  ◦ Adding to BuildConfig ...................................................... 2209
• **Advanced Emulator Capabilities**
  ◦ Prerequisites ................................................................... 2213
  ◦ x86 Images ..................................................................... 2213
  ◦ Hardware Graphics Acceleration ....................................... 2216
  ◦ Defining New Devices ....................................................... 2219
  ◦ Keyboard Behavior .......................................................... 2222
  ◦ Headless Operation .......................................................... 2222
• **Using Lint**
  ◦ Prerequisites ................................................................... 2223
  ◦ What It Is ......................................................................... 2223
  ◦ When It Runs ................................................................... 2224
  ◦ What to Fix ....................................................................... 2226
  ◦ What to Configure .......................................................... 2226
• **Using Hierarchy View**  
  ○ Prerequisites ................................................................. 2231  
  ○ Launching Hierarchy View ............................................... 2231  
  ○ Viewing the View Hierarchy ........................................... 2232  
  ○ ViewServer .................................................................. 2235

• **Using DDMS**  
  ○ Prerequisites ................................................................... 2237  
  ○ Starting DDMS .............................................................. 2237  
  ○ File Push and Pull ........................................................... 2238  
  ○ Screenshots .................................................................... 2239  
  ○ Location Updates ........................................................... 2239  
  ○ Placing Calls and Messages ............................................ 2240

• **Android Development in IntelliJ IDEA**  
  ○ Prerequisites .................................................................... 2243  
  ○ Creating a New Project ................................................... 2243  
  ○ Importing an Existing Project ........................................... 2246  
  ○ Attaching a JAR ................................................................ 2256  
  ○ Accessing Android Tools ............................................... 2258  
  ○ Run and Debug a Project .................................................. 2260  
  ○ Editing Android-Specific Files ........................................ 2261  
  ○ IDEA-Specific Files .......................................................... 2263  
  ○ Using MAT ...................................................................... 2263

• **Signing Your App**  
  ○ Prerequisites .................................................................... 2265  
  ○ Role of Code Signing ...................................................... 2265  
  ○ What Happens In Debug Mode ........................................ 2266  
  ○ Creating a Production Signing Key ................................... 2266

• **Distribution**  
  ○ Prerequisites .................................................................... 2273  
  ○ Get Ready To Go To Market ............................................. 2273

• **Issues with Speed**  
  ○ Prerequisites .................................................................... 2279  
  ○ Getting Things Done ...................................................... 2279  
  ○ Your UI Seems... Janky .................................................... 2280  
  ○ Not Far Enough in the Background ................................... 2280  
  ○ Playing with Speed .......................................................... 2281

• **Finding CPU Bottlenecks**  
  ○ Prerequisites .................................................................... 2283  
  ○ Traceview ....................................................................... 2284  
  ○ Other General CPU Measurement Techniques ................ 2293  
  ○ **UI “Jank” Measurement** ............................................... 2295
• **Focus On: NDK**
  - Prerequisites ................................................................. 2311
  - The Role of the NDK ...................................................... 2312
  - NDK Installation and Project Setup .................................. 2315
  - Writing Your Makefile(s) ............................................... 2319
  - Building Your Library .................................................. 2320
  - Using Your Library Via JNI ............................................ 2321
  - Building and Deploying Your Project ............................... 2326

• **Improving CPU Performance in Java**
  - Prerequisites ................................................................ 2329
  - Reduce CPU Utilization .................................................. 2329
  - Reduce Time on the Main Application Thread ................... 2334
  - Improve Throughput and Responsiveness ......................... 2342

• **Finding and Eliminating Jank**
  - Prerequisites ................................................................ 2345
  - The Case: ThreePaneDemoBC .......................................... 2345
  - Are We Janky? .............................................................. 2346
  - Finding the Source of the Jank ....................................... 2346
  - Where Things Went Wrong ............................................ 2356
  - Removing the Jank ....................................................... 2357

• **Issues with Bandwidth**
  - Prerequisites ................................................................ 2359
  - You're Using Too Much of the Slow Stuff ....................... 2360
  - You're Using Too Much of the Expensive Stuff ............... 2360
  - You're Using Too Much of Somebody Else's Stuff .......... 2361
  - You're Using Too Much... And There Is None ................ 2362

• **Focus On: TrafficStats**
  - Prerequisites ................................................................ 2363
  - TrafficStats Basics ....................................................... 2363
  - Example: TrafficMonitor ................................................ 2365
  - Other Ways to Employ TrafficStats ................................ 2373

• **Measuring Bandwidth Consumption**
  - Prerequisites ................................................................ 2375
  - On-Device Measurement ................................................ 2375
  - Off-Device Measurement ................................................. 2377
  - Tactical Measurement in DDMS ..................................... 2379

• **Being Smarter About Bandwidth**
  - Prerequisites ................................................................ 2383
  - Bandwidth Savings ....................................................... 2383
  - Bandwidth Shaping ....................................................... 2389
  - Avoiding Metered Connections ..................................... 2392
• **Issues with Application Heap**
  - Prerequisites ................................................................. 2395
  - *You Are in a Heap of Trouble* ........................................ 2396
  - *Determining Your Heap Size At Runtime* ......................... 2396
  - *Fragments of Memory* .................................................. 2397
  - *Getting a Trim* ............................................................. 2398
  - *Warning: Contains Graphic Images* ................................. 2399
  - *Releasing SQLite Memory* ............................................. 2402
  - *In Too Deep (on the Stack)* ........................................... 2402

• **Finding Memory Leaks with MAT**
  - Prerequisites ........................................................................ 2405
  - Setting Up MAT ..................................................................... 2405
  - Getting Heap Dumps .......................................................... 2406
  - Basic MAT Operation .......................................................... 2412
  - Some Leaks and Their MAT Analysis .................................... 2418

• **Issues with System RAM**
  - Prerequisites ........................................................................ 2427
  - *Can’t We All Just Get Along?* ........................................... 2427
  - *Contributors to System RAM Consumption* ....................... 2428
  - *Measuring System RAM Consumption: Tools* ...................(tools) 2429
  - *Measuring System RAM Consumption: Runtime* ............... 2444
  - *Learn To Let Go (Of Your Heap)* ...................................... 2446

• **Issues with Battery Life**
  - Prerequisites ........................................................................ 2447
  - You’re Getting Blamed ....................................................... 2448
  - Not All Batteries Are Created Equal .................................... 2449
  - Stretching Out the Last mWWh ........................................... 2449

• **Power Measurement Options**
  - Prerequisites ........................................................................ 2451
  - The Qualcomm Tool (That Must Not Be Named) .................. 2452
  - PowerTutor .......................................................................... 2453
  - *Battery Screen in Settings Application* ............................. 2457
  - BatteryInfo Dump ............................................................. 2459

• **Sources of Power Drain**
  - Prerequisites ........................................................................ 2463
  - Screen .................................................................................. 2464
  - Disk I/O ................................................................................ 2465
  - WiFi and Mobile Data .......................................................... 2466
  - GPS ...................................................................................... 2469
  - Camera .................................................................................. 2470
  - Additional Sources ............................................................. 2470
• **Addressing Application Size Issues**
  - Prerequisites ................................................................. 2473
  - *Java Code, and the 64K Method Limit* .......................... 2473
  - Native Code ................................................................. 2477
  - Images ............................................................................. 2479
  - APK Expansion Files ...................................................... 2481
• **The Role of Alternative Environments**
  - Prerequisites ................................................................. 2483
  - In the Beginning, There Was Java.................................. 2484
  - ... And It Was OK .......................................................... 2484
  - Bucking the Trend .......................................................... 2485
  - Support, Structure .......................................................... 2485
  - Caveat Developer ............................................................. 2486
• **HTML5**
  - Prerequisites ................................................................. 2487
  - Offline Applications ...................................................... 2487
  - Web Storage ..................................................................... 2494
  - *Going To Production* ..................................................... 2497
  - Issues You May Encounter ............................................. 2498
  - HTML5: The Baseline ...................................................... 2501
• **PhoneGap**
  - Prerequisites ................................................................. 2503
  - What Is PhoneGap? ......................................................... 2503
  - Using PhoneGap ............................................................. 2506
  - PhoneGap and the Checklist Sample ............................... 2511
  - Issues You May Encounter ............................................. 2516
  - For More Information .................................................. 2519
• **Other Alternative Environments**
  - Prerequisites ................................................................. 2521
  - Rhodes ............................................................................. 2521
  - Flash, Flex, and AIR ....................................................... 2522
  - JRuby and Ruboto ............................................................ 2522
  - App Inventor ..................................................................... 2523
  - Titanium Mobile ............................................................. 2525
  - Other JVM Compiled Languages .................................... 2526
• **Anti-Patterns**
  - Prerequisites ................................................................. 2529
  - Leak Threads... Or Things Attached to Threads .................. 2529
  - Use Large Heap Unnecessarily ........................................ 2531
  - Misuse the MENU Button .............................................. 2533
  - Interfere with Navigation .............................................. 2534
• Use android:sharedUserId ......................................................... 2536
• Implement a “Quit” Button ......................................................... 2537
• Terminate Your Process ............................................................. 2539
• Try to Hide from the User .......................................................... 2540
• Use Multiple Processes ............................................................. 2541
• Hog System Resources ............................................................. 2543
  • Widget Catalog: AdapterViewFlipper  
    ◦ Key Usage Tips ................................................................. 2545
    ◦ A Sample Usage .............................................................. 2546
    ◦ Visual Representation ....................................................... 2546
  • Widget Catalog: CalendarView  
    ◦ Key Usage Tips ................................................................. 2547
    ◦ A Sample Usage .............................................................. 2548
    ◦ Visual Representation ....................................................... 2549
  • Widget Catalog: DatePicker  
    ◦ Key Usage Tips ................................................................. 2551
    ◦ A Sample Usage .............................................................. 2551
    ◦ Visual Representation ....................................................... 2553
  • Widget Catalog: ExpandableListView  
    ◦ Key Usage Tips ................................................................. 2557
    ◦ A Sample Usage .............................................................. 2558
    ◦ Visual Representation ....................................................... 2564
  • Widget Catalog: SeekBar  
    ◦ Key Usage Tips ................................................................. 2567
    ◦ A Sample Usage .............................................................. 2567
    ◦ Visual Representation ....................................................... 2569
  • Widget Catalog: SlidingDrawer  
    ◦ Key Usage Tips ................................................................. 2571
    ◦ A Sample Usage .............................................................. 2572
    ◦ Visual Representation ....................................................... 2573
  • Widget Catalog: StackView  
    ◦ Key Usage Tips ................................................................. 2575
    ◦ A Sample Usage .............................................................. 2576
    ◦ Visual Representation ....................................................... 2577
  • Widget Catalog: TabHost and TabWidget  
    ◦ Deprecation Notes ............................................................. 2579
    ◦ Key Usage Tips ................................................................. 2579
    ◦ A Sample Usage .............................................................. 2580
    ◦ Visual Representation ....................................................... 2582
  • Widget Catalog: TimePicker  
    ◦ Key Usage Tips ................................................................. 2585
- A Sample Usage ................................................................. 2585
- Visual Representation ...................................................... 2587

• Widget Catalog: ViewFlipper
  - Key Usage Tips ............................................................. 2589
  - A Sample Usage ........................................................... 2590
  - Visual Representation .................................................... 2591

• Device Catalog: Kindle Fire
  - Prerequisites ..................................................................... 2593
  - Introducing the Kindle Fire series ...................................... 2593
  - What Features and Configurations Does It Use? ................. 2594
  - What Is Really Different? .................................................. 2596
  - Getting Your Development Environment Established .......... 2602
  - How Does Distribution Work? .......................................... 2604
  - Amazon Equivalents of Google Services .............................. 2605
  - Getting Help with the Kindle Fire ...................................... 2606

• Nokia X Series
  - Prerequisites .................................................................... 2608
  - What Is Really Different? ................................................... 2608
  - Getting Your Development Environment Established .......... 2609
  - Testing Your App ............................................................. 2611
  - Nokia X Analyzer ............................................................ 2612
  - Your Play Services Alternative is HERE (In Part) ................. 2614
  - How Does Distribution Work? .......................................... 2614
  - Getting Help ................................................................. 2615

• Device Catalog: Barnes & Noble NOOK Tablet
  - Prerequisites ..................................................................... 2617
  - What Features and Configurations Does It Use? ................. 2618
  - What Is Really Different? ................................................... 2618
  - Getting Your Development Environment Established .......... 2620
  - How Does Distribution Work? .......................................... 2622

• Device Catalog: RIM Blackberry Playbook
  - What Features and Configurations Does It Use? ................. 2623
  - What Is Really Different? ................................................... 2624
  - Getting Your Development Environment Established .......... 2625
  - How Does Distribution Work? .......................................... 2627

• Wrist Wearables
  - Prerequisites ..................................................................... 2630
  - Divvying Up the Wearables Space ...................................... 2630
  - Example Wrist Wearables .................................................. 2631
  - Strategic Considerations .................................................. 2634
  - Tactical Considerations .................................................... 2636

xxxii
What About Android Wear? ........................................................... 2638

- **Device Catalog: Google TV**
  - Prerequisites ........................................................................ 2641
  - What Features and Configurations Does It Use? ..................... 2642
  - **What Is Really Different?** ............................................... 2643
  - Getting Your Development Environment Established ............ 2646
  - How Does Distribution Work? ............................................. 2649
  - Getting Help ........................................................................ 2651

- **Amazon Fire TV**
  - Prerequisites ........................................................................ 2653
  - Introducing the Fire TV .......................................................... 2653
  - What Features and Configurations Does It Use? ..................... 2658
  - What Is Really Different? ....................................................... 2660
  - Casting and Fire TV ............................................................... 2660
  - Getting Your Development Environment Established ............ 2661
  - Working with the Remote and Controller .............................. 2663
  - How Does Distribution Work? ............................................. 2665
  - Getting Help ........................................................................ 2665

- **CWAC Libraries**
  - cwac-adapter ........................................................................ 2667
  - cwac-camera ......................................................................... 2667
  - cwac-colormixer .................................................................... 2668
  - cwac-layouts .......................................................................... 2668
  - cwac-merge ........................................................................... 2668
  - cwac-pager ............................................................................. 2669
  - cwac-presentation .................................................................. 2669
  - cwac-provider ........................................................................ 2669
  - cwac-richedit ......................................................................... 2669
  - cwac-sacklist ......................................................................... 2669
  - cwac-security ......................................................................... 2670
  - cwac-strictmodeex ............................................................... 2670
  - cwac-wakeful ......................................................................... 2670
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

Once upon a time, CommonsWare published a few books on Android development. What you are reading represents the merger of those separate titles into a single omnibus title.

To make the equivalent of 2,000+ 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 (in the style of CommonsWare’s former *Android Programming Tutorials*), to give you hands-on experience with the concepts being discussed. Most of the tutorials can be skipped, 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):

**Advanced UI**

- Introducing GridLayout
- Dialogs and DialogFragments
- Advanced ListViews
- Action Bar Navigation
- Action Modes and Context Menus
- ActionBarCompat: The Official Action Bar Backport
- Other Advanced Action Bar Techniques
• Implementing a Navigation Drawer
• Advanced Uses of WebView
• The Input Method Framework
• Fonts
• Rich Text
• Custom Drawables
• Animators
• Legacy Animations
• 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 Buses

Home Screen Effects

• Home Screen App Widgets
• Adapter-Based App Widgets

Data Storage and Retrieval

• Content Provider Theory
• Content Provider Implementation Patterns
• The Loader Framework
• The ContactsContract Provider
• The CalendarContract Provider
• The MediaStore Provider
• Encrypted Storage
• Packaging and Distributing Data

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”

Security

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

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
• Push Notifications with GCM
• Basic Use of Sensors
• Other System Settings and Services
• Dealing with Different Hardware

Integration and Introspection

• Responding to URLs
• Plugin Patterns
• PackageManager Tricks
• Searching with SearchManager
• System Events
• 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

Testing

• JUnit and Android
• MonkeyRunner and the Test Monkey
• Testing with UIAutomator

Gradle and the New Build System

• Introducing Gradle
• Gradle and Legacy Projects
• Gradle and the New Project Structure
• Gradle and Dependencies
• Gradle and Testing
• Advanced Gradle for Android Tips

Other Tools

• Advanced Emulator Capabilities
• Using Lint
• Using Hierarchy View
• Using DDMS
• Finding CPU Bottlenecks
• Finding Memory Leaks with MAT
• Android Development with IntelliJ IDEA

Production

• Signing Your App
• Distribution
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
- SlidingDrawer
- StackView
- TabHost
**Device Catalog**

- Kindle Fire
- Nokia X Series
- Barnes & Noble NOOK Tablet
- RIM Blackberry Playbook
- 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 other titles that CommonsWare may publish during that subscription period.

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

Subscribers also have access to “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.

**Getting Help**

If you have questions about the book examples, visit StackOverflow and ask a question, tagged with android and commonsware.

If you have general Android developer questions, visit StackOverflow and ask a question, tagged with android (and any other relevant tags, such as java).

**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
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 CommonsWare Web site, bear in mind that the projects are set up to be built by Eclipse. Many are also set up to be built by Ant from the command line. However, for command-line builds, you will need to update the build files to match your local environment. To do this, delete build.xml in your project directory, then run android update project -p. from that same directory. See the GitHub repo home page for more details.

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.

**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 April 2018. 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 StackOverflow.

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.
Core Chapters
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.
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.
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:

```java
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 enumerated 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.

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.
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 short 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.
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:

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

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 100 million Android devices in use today, representing hundreds 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 three dominant “form factors”:

- the phone
- the tablet
- the television (TV)

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 “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?
KEY ANDROID CONCEPTS

- does the device have telephony capability?
- etc.

Similarly, as you build your applications, rather than thinking of those three 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:

- watches and other types of wearable devices
- airplane seat-back entertainment centers
- in-car navigation and entertainment devices
- and so on

The Emulator

While there are hundreds of millions of Android devices representing hundreds of models, you probably 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 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)

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 is 19, representing Android 4.4.

**Dalvik**

You probably are thinking that Dalvik is a village in Iceland. That, however, is 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.
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.

**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.
Choosing Your IDE

Before you go much further in your Android endeavors (or, possibly, endeavours, depending upon your preferred spelling), you will need to determine what tools you will use to build your Android applications. Many developers are used to using an integrated development environment (IDE). Android has excellent support for Eclipse, and other IDEs offer varying degrees of Android integration. You do not necessarily have to use an IDE, though, if you do not wish to.

This chapter will outline your options in this area.

Eclipse

Eclipse is an extremely popular IDE, particularly for Java development. It is also designed to be extensible via an add-in system. To top it off, Eclipse is open source. That combination made it an ideal choice of IDE to get attention from the core Android developer team.

Specifically, to go alongside the Android SDK, Google has published some add-ins for the Eclipse environment. Primary among these is the Android Developer Tools (ADT) add-in, which gives the core of Eclipse awareness of Android.

What the ADT Gives You

The ADT add-in, in essence, takes regular Eclipse operations and extends them to work with Android projects. For example, with Eclipse, you get:

- New project wizards to create regular Android projects, Android test projects, etc.
The ability to run an Android project just like you might run a regular Java application — via the green “run” button in the toolbar — despite the fact that this really involves pushing the Android application over to an emulator or device, possibly even starting up the emulator if it is not running.

• Tooltip support for Android classes and methods

Eclipse and the ADT also offer preliminary support for drag-and-drop GUI editing. While this book will also cover the XML files that Eclipse will generate, Eclipse now lets you assemble those XML files by dragging UI components around on the screen, adjusting properties as you go.

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.

Out of all the shortcut key-combinations for Eclipse, two of the most important for readers of this book, particularly if you are following the tutorials, are:

• <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
• <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.

Android Studio

The next-generation IDE from Google is Android Studio, based upon IntelliJ IDEA. This is still in an “early-access preview” state today and is not recommended for newcomers to Android development. It should stabilize to a 1.0 (or equivalent) release in 2014, at which time this book will start to cover it in detail.

Alternative IDEs

Other IDEs are slowly getting their equivalents of the ADT, albeit with minimal assistance from Google. For example, IntelliJ IDEA has a module for Android – originally commercial, it is part of the open source community edition of IDEA as of version 10. Also, NetBeans has support via the NBAndroid add-on, and reportedly this has advanced substantially in the past few years.
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.

**IDEs… And This Book**

You are welcome to use 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 Eclipse-specific instructions, since it is so popular.

**About App Inventor**

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

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, who has recently re-opened it [to the public](https://appinventor.mit.edu/explore/appinventor/). 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.

This book does not cover the use of App Inventor.
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: Eclipse and the Android emulator. Of the two, the emulator poses the bigger problem.

The more RAM you have, the better. 3GB or higher is a very good idea if you intend to use Eclipse 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. Smaller screens (e.g., phones) can run acceptably on 2.5GHz and (slightly) slower CPUs.
Step #2 - Setting Up Java

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).

Hence, the first thing you need to do is get set up with a Java development environment and be ready to start writing Java classes.

Install the JDK

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 and Linux, and presumably from Apple for OS X. 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 the latter will require you to configure your IDE to compile your Java code to Java 6 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).

Step #3 - Install the Android SDK

The Android SDK 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.

Install the Base Tools

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

The default option at present is for you to download the “ADT Bundle”. This includes a complete copy of Eclipse, along with the base tools and the latest SDK files. If you want a temporary Android development environment, this is probably a fine choice.

Otherwise, 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
TUTORIAL #1 - INSTALLING THE TOOLS

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.

Install the SDKs and Add-Ons

Inside the tools/ directory of your Android SDK installation from the previous step, you will see an android batch file or shell script. On Windows, you may see an “SDK Manager.exe” file, perhaps in the root of your ADT bundle installation. In either case, if you run that, you will be presented with the Android SDK Manager:

![Android SDK Manager](image)

Figure 3: 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 check the following items:

1. “SDK Platform” for all Android SDK releases you want to test against — for this book API 15 (Android 4.0.3) is recommended, along with any others with which you wish to experiment.
TUTORIAL #1 - INSTALLING THE TOOLS

2. “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, though setting that up is a bit of an advanced topic.
4. “Samples for SDK” for the latest Android SDK release, and perhaps for older releases if you wish.
5. “Google APIs” for each Android SDK release for which you are downloading the platform (see first bullet).
6. Android SDK Tools and Platform-tools.
7. Android Support Library (in the Extras group at the bottom of the tree).

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

![Android SDK Manager Installing Packages]

Review and accept the licenses, then click the Install button. At this point, this is a fine time to go get lunch. Or, perhaps dinner. Unless you have a substantial Internet connection, downloading all of this data and unpacking it will take a fair bit of time.

When the download is complete, you can close up the SDK Manager if you wish, though we will use it to set up the emulator in a later step of this chapter.

**Step #4 - Install the ADT for Eclipse**

If you will not be using Eclipse for your Android development, you can skip to the next section. Similarly, if you downloaded the “ADT Bundle” and therefore already have a completely-configured Eclipse environment, you can skip to the next section.
TUTORIAL #1 - INSTALLING THE TOOLS

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 3.6 (Helios) 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:

![Eclipse ADT plug-in installation](image)

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:

![Preferences window](image)

*Figure 6: 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 #5 - Install Apache Ant**

If you will be doing all of your development from Eclipse, you can skip to the next section.
TUTORIAL #1 - INSTALLING THE TOOLS

If you wish to develop using command-line build tools, you will need to install Apache Ant. You may have this already from previous Java development work, as it is fairly common in Java projects. However, you will need Ant version 1.8.1 or higher, so double-check your current copy (e.g., `ant -version`) to ensure you are on the proper edition.

If you do not have Ant, you can obtain it from the Apache Ant Web site. They have full installation instructions in the Ant manual, but the basic steps are:

- Unpack the ZIP archive wherever it may make sense on your machine
- Add a JAVA_HOME environment variable, pointing to where your JDK is installed, if you do not have one already
- Add an ANT_HOME environment variable, pointing to the directory where you unpacked Ant in the first step above
- Add $JAVA_HOME/bin and $ANT_HOME/bin to your PATH (note: Windows users would add %JAVA_HOME%\bin and %ANT_HOME%\bin)
- Run ant -version to confirm that Ant is installed properly

Step #6 - 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.

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. From the command line, you can bring up the AVD Manager via the `android avd` command from your SDK’s tools/ directory. From Eclipse, you start the AVD Manager via its toolbar button or via the Window | AVD Manager main menu option. It starts up on a screen listing the AVDs you have available – initially, the list will be empty.
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 8: AVD Manager, Device Definitions Tab
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 9: Adding a New AVD](image)

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 — it is a fairly common resolution that will not be too terribly slow.
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.
4. The CPU architecture your emulator will emulate. The vast majority of Android devices have ARM CPUs, while the vast majority of development machines have x86 CPUs. However, since setting up the x86 emulator support is a bit complicated, for now, choose ARM. Later on, though, you really will want to consider an x86 emulator, as they tend to run much faster.

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+):
... then a graphical Android logo:
 Tutorial #1 - Installing the Tools

Figure 11: Android emulator, secondary startup segment

before eventually landing at the home screen, a welcome page (shown below, for Android 4.0), or the keyguard:
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:
Step #7 - 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
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).

![Developer options](image)

*Figure 14: Android 4.0 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.
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.

**Manufacturer-Supplied Driver**

If you still do not have a driver, the [OEM USB Drivers](https://commonsware.com) 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...
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, Android 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.

In Our Next Episode…

... we will [create an Android project](https://commonsware.com) that will serve as the basis for all our future
tutorials.
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.
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. You can import these projects into Eclipse, 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.

All that being said, it is a pretty good idea to do at least this tutorial, so you learn how to create an Android project.

About the Eclipse Instructions

The instructions found in this book assume that you are using the R22.6 version of the Android developer tools and the ADT plugin for Eclipse. Newer or older versions may have somewhat different details, though most of what you see here should be applicable.
Note that the R22.6 version of the Eclipse new-project wizard is fairly broken. It generates projects that do not work, in part because the wizard templates add in some external dependencies that the wizard does not set up. And, we will not be covering those dependencies for quite some time in the book.

As a result, the Eclipse instructions for Step #1 come in two forms:

- The first walks you through how to use the new-project wizard... before throwing out the results
- The second shows you how to import an Android project, where we will choose a fairly blank project as the starting point

**Step #1: Creating the Project**

First, we need to create the Android project for EmPubLite. You need to decide whether you are going to work with this project from inside the Eclipse IDE or through other tools.

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 three sets of instructions:

1. Using the Eclipse new-project wizard (not recommended)
2. Importing a blank project into Eclipse
3. Using command-line tools, for use outside of Eclipse

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

**Eclipse New-Project Wizard**

NOTE: The instructions shown in this section will walk you through the Eclipse new-project wizard. As was noted above, this wizard is presently rather broken. You will not use the results of these instructions. If you want to skip this part, feel free to jump ahead to the “Eclipse Project Import” instructions, which are the “real” Eclipse instructions until the new-project wizard works better.

From the Eclipse main menu, choose File > New > Project... to bring up the first page of the “New Project” wizard:
Choose “Android Application Project” from the types of projects and click “Next >” to proceed to the next page of the wizard:
Fill in the following items:

- For “Application Name” and “Project Name”, fill in EmPubLite
- For “Package Name”, fill in com.commonsware.empublite
- For “Minimum Required SDK”, choose “API 9: Android 2.3 (Gingerbread)”
- For “Target SDK”, choose “API 15: Android 4.0.3 (IceCreamSandwich)”
- For “Compile With”, choose “API 15: Android 4.0.3 (IceCreamSandwich)” (if you do not have that version, any higher API level should be fine)

The remaining defaults should be fine, leaving you with a dialog akin to this:
Then, click “Next >” to move to the next page of the wizard:
Here:

- Uncheck “Create custom launcher icon”, as we will do this separately later
- Leave “Create activity” checked
- Leave “Mark this project as a library” unchecked
- 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)
Then, click “Next >” to move to the next page of the wizard:
Here, you choose which template project you want to use as a starting point. Leave the “Create Activity” checkbox checked, and choose “BlankActivity” from the template list.

Then, click “Next >” to move to the next page of the wizard:
Fill in the following details:

- For “Activity Name”, fill in EmPubLiteActivity
- For “Layout Name”, fill in main
- For “Fragment Layout Name”, fill in blank_means_blank_plz

Leave the rest of the defaults alone.
At this point, you can click the “Finish” button to complete the wizard. Your new EmPubLite project should appear in the Eclipse Package Explorer view:
As noted previously, the project starts off in a broken state, due to bugs in Eclipse. It also adds a lot of extra stuff that we do not need or want for this particular project. As such, these instructions were, unfortunately, useless.

Right-click over the EmPubLite project name in the Package Explorer, and choose New > Delete from the context menu that appears. In the resulting dialog, check the “Delete project contents on disk (cannot be undone)” checkbox, then click OK to delete the project. Then, continue on with the instructions in the next section, to import a blank Eclipse project.

**Eclipse Project Import**

First, visit the book’s GitHub repository’s “releases” area and download the BlankProject.zip file that corresponds with this version of the book. If there is none for this version of the book, work your way backwards to older versions until you find one.

Then, unZIP that ZIP archive into some directory on your development machine. 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.
Then, in Eclipse, choose File > Import from the main menu, to bring up the import wizard:

![Eclipse Import Wizard, First Page](image)

Choose “Existing Android Code Into Workspace” and click the Next button to bring up the next page of the wizard:
Click the “Browse...” button towards the upper-right of the wizard, and from there choose the directory containing the unZIpped version of BlankProject.zip.

This will change the wizard to show the project:
Click on “MainActivity” in the table, which will then become editable. Change MainActivity to be EmPubLite:
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:
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:
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.

We will need to do some additional work to make this imported project look more like the one we should have gotten from the new-project wizard, but we will handle that work in later tutorials.

Note that you can skip over the next section and continue with Step 2, if you do not want to experiment with creating a new project from the command line.

**Command Line**

First, choose where you want to create the project on your filesystem.

Then, execute the following command:

```
android create project -n EmPubLite -t android-15 -p ... -k com.commonsware.empublite -a EmPubLiteActivity
```

(replacing the ... with the path to your desired project directory)
This will:

- Create the directory you specified
- Create a bunch of files in that directory, using the package name and activity name that you supplied

If `android create project` is not recognized as a command, be sure that you added your SDK’s `tools/` and `platform-tools/` directories to your `PATH` environment variable (and restarted your command line, if needed).

If you get a complaint that you do not have `android-15`, either use the SDK Manager to install the SDK for Android’s API Level 15 (Android 4.0.3) or adjust that parameter to some SDK that you have (e.g., `android-18`).

**Step #2: 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 Eclipse versus command-line development — please follow the instructions that are appropriate for you.

**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:
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. Otherwise, Eclipse will start up a suitable emulator, from the AVDs you created in the previous tutorial, then will install and run the app on it:
Note that you will have to unlock your device or emulator to actually see the app running — it will not unlock automatically for you.

**Command Line**

First, you need to either attach a device or start up a 4.0.3 emulator (we will add support for earlier versions of Android in an upcoming tutorial). If you did not create a 4.0.3 AVD in the first tutorial, and you do not have an Android device running 4.0.3 or higher, go ahead and create the 4.0.3 emulator AVD.

To start the emulator, execute the `android avd` command to bring up the AVD Manager:

![Android AVD Manager](image)

*Figure 33: Android AVD Manager*

Highlight the AVD you wish to run, then click “Start...”:
You can, if you wish, just click “Launch” to start up the emulator. Or, you can tailor the output, such as by checking the “Scale display to real size” checkbox, then filling in the desired diagonal size of the emulator screen and the dots-per-inch (dpi) of your development machine’s monitor. Clicking the “?” will bring up an assistant that will help you calculate your monitor’s dots-per-inch.

Once your emulator is launched, from your project directory, run the `ant clean debug install` command. This will:

- Clean out any pre-compiled stuff from previous builds
- Create a debug build of your app
- Install that debug build on your emulator

If you navigate to the launcher of the emulator, you will see your EmPubLite icon — tapping that will bring up the do-nothing stub application.

**In Our Next Episode…**

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

### Root Contents

When you create a new Android project (e.g., via `android create project`), you get several items in the project's root directory, including:

1. `AndroidManifest.xml`, which is an XML file describing the application being built and what components — activities, services, etc. — are being supplied by that application
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 “resources”, such as icons, GUI layouts, and the like, that get packaged with the compiled Java in the application
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
   (NOTE: this directory may not be created for you by Eclipse, though it is by
   the command-line option, and you can add it yourself to your Eclipse project
   when needed)
4. `build.xml` and `*.properties`, which are used as part of the Ant-based
   command-line build process, if you are not using Eclipse
5. `proguard.cfg` or `proguard-project.txt`, which are used for integration with
   ProGuard for obfuscating your Android code
6. Eclipse project files (e.g., `.classpath`), if you are using Eclipse

The Sweat Off Your Brow

When you created the project (e.g., via `android create 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 `src/`
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 others to the
`src/` tree as needed to implement your application, and we will demonstrate that
countless times as we progress through this book.

The first time you compile the project (e.g., via `ant`), out in the project’s package’s
directory, the Android build chain will create `R.java`. This contains a number of
constants tied to the various resources you placed out in the `res/` directory tree. You
should not modify `R.java` yourself, letting the Android tools handle it for you. You
will see throughout many of the samples where we reference things in `R.java` (e.g.,
referring to a layout’s identifier via `R.layout.main`).

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.

**What You Get Out Of It**

When you compile your project (via ant or the IDE), the results go into the bin/ directory under your project root. Specifically:

1. bin/classes/ holds the compiled Java classes
2. bin/classes.dex holds the executable created from those compiled Java classes
3. bin/yourapp.ap_ holds your application’s resources, packaged as a ZIP file (where yourapp is the name of your application)
4. bin/yourapp-*.apk is the actual Android application (where * varies)

The .apk file is a ZIP archive containing the .dex file, 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.
Inside the Manifest

The foundation for any Android application is the manifest file: AndroidManifest.xml in the root of your project. 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 API demo suite is over 1,000 lines long. Your production Android applications will probably fall somewhere in the middle.

In The Beginning, There Was the Root, And It Was Good

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

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.commonsware.cwac.richedit.demo"
android:versionCode="1"
android:versionName="1.0">
```

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. Here, you can provide the name of the Java package that will be considered the “base” of your application. Your package is a unique identifier for
your application. A device can only have one application installed with a given package, and the Play Store will only list one project with a given package.

Your manifest also specifies android:versionName and android:versionCode attributes. These represent the versions of your application. The android:versionName value is what the user will see for a version indicator in the Applications details screen for your app in their Settings application:

![Image of app version in Settings]

*Figure 35: 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 android: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.
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
<?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>
```

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.
Specifying Versions

As was noted earlier in this chapter, your manifest already contains some version information, about your own application’s version. It also contains a `<uses-sdk>` element as a child of the `<manifest>` element to your AndroidManifest.xml file, to specify what versions of Android you are supporting.

The most important attribute for your `<uses-sdk>` element is `android: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 2.1 and newer versions of Android, you would set your `android:minSdkVersion` to be 7.

You should also specify an `android:targetSdkVersion` attribute. 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. In particular, to get the new “Honeycomb” look-and-feel when running on an Android 3.0 (or higher) device, you need to specify a target SDK version of 11 or higher:

```xml
<uses-sdk android:minSdkVersion="7" android:targetSdkVersion="11" /> 
```

Supporting Multiple Screens

Android devices come with a wide range of screen sizes, from 2.8” tiny smartphones to 46” Google 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.

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
- `<uses-library>`, to tell Android that you need the device to support a certain library in firmware (e.g., Google Maps), and therefore your app should not be installed on devices lacking that library

These and other elements will be introduced elsewhere in the book.
Tutorial #3 - Changing Our Manifest

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.

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.

Step #1: Updating the Package Name

If you imported the blank project in Eclipse as your starting point in the previous tutorial, that project will have a different package name than what we want to use. If, on the other hand, you used the Eclipse new-project wizard or command-line android create project commands to create your starting point, your package name is already set, and you can skip ahead to the next section.

Double-click on the AndroidManifest.xml file for your EmPubLite project in the Eclipse Package Explorer. 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:
In the “Package” field towards the top, replace `com.commonsware.android.blank` with `com.commonsware.empublite`. Then, save your changes, via the “save” icon in the Eclipse toolbar, via `<Ctrl>-<S>`, etc. If a dialog appears asking if you want to update your launch configurations, click Yes.

At this point, you should have some red X marks over your source files, as those files are expecting to be in the same package as is used for the overall application. To fix this, right-click over the `com.commonsware.android.blank` package in the `src` folder in Eclipse's Package Explorer, then choose Refactor > Rename from the context menu, to bring up the Rename Package dialog:
Change the contents of the “New name” field to be `com.commonsware.empublite`. Leave everything else alone, and click OK. If a “found problems” dialog appears, click Continue.

**Step #2: 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).

If you wish to make this change using Eclipse’s structured manifest editor, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**Eclipse**

If you skipped over Step #1, then in the Package Explorer view in Eclipse, find the `AndroidManifest.xml` entry and double-click on it.
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:

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:

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:
TUTORIAL #3 - CHANGING OUR MANIFEST

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 42: Eclipse Supports Screens Options, Adjusted](image)

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

Outside of Eclipse

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

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

Step #3: Validating our Minimum and Target SDK Versions

If you created your project using the Eclipse new-project wizard, then in the “Manifest Extras” area of the “Manifest” sub-tab in our open manifest editor, you should have a Uses Sdk entry. Clicking on that should show that your minimum SDK version is set to 9 and that your target SDK version is 15 (or whatever you chose in Tutorial #2):
If you created your project via importing the blank project into Eclipse, your values will be 15 for the minimum SDK version and 19 for the target SDK version, respectively. Change them to match the 9 and 15 shown above.

If you created your project from the command line, this element may not exist. You will need to add a `<uses-sdk android:minSdkVersion="9" android:targetSdkVersion="15"/>` element to your manifest, as a child of the root `<manifest>` element.

The entire manifest file, at this point should look a bit like:

```xml
<?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">

    <uses-sdk
        android:minSdkVersion="9"
        android:targetSdkVersion="15"/>

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

    <application
        android:allowBackup="true"
        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"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```
<manifest>

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

In Our Next Episode…

... we will make some changes to the resources of our tutorial project
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. 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.

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:

```xml
<resources>
  <string name="quick">The quick brown fox...</string>
  <string name="laughs">He who laughs last...</string>
</resources>
```

The only 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, our stub project's strings.xml file looks like this:

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">EmPubLite</string>
  <string name="hello_world">Hello world!</string>
  <string name="menu_settings">Settings</string>
</resources>
```

We will reference these string resources from various locations, in our Java source code and elsewhere. For example, the app_name string resource is used in our AndroidManifest.xml file:

```xml
<?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">
  ...
  <uses-sdk
    android:minSdkVersion="9"
    android:targetSdkVersion="15"/>
  ...
  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true">
  </supports-screens>
</manifest>
```
Here, the `android:label` attribute of our `<application>` element refers to the `app_name` string resource. This will appear in a few places in our 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, such as `<b>`, `<i>`, and `<u>`. Your string resources support this, simply by using the HTML tags as you would in a Web page:

```xml
<resources>
  <string name="b">This has <b>bold</b> in it.</string>
  <string name="i">Whereas this has <i>italics</i>!</i></string>
</resources>
```

Unfortunately, the list of supported tags is undocumented. Based on recent Android implementations, it will mostly be your inline markup rules (e.g., `<tt>`, `<h1>`, `<small>`, `<strike>`).
The Directory Name

Our string resources in our stub project are in the res/values/strings.xml file. This directory (res/values/) means that 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.

String Resources and Eclipse

When you double-click on a string resource file, like res/values/strings.xml, you will be greeted with a list of all the string resources that have been defined:

![Figure 44: Eclipse, Showing String Resources](image)

Clicking on a resource allows you to edit its name and value:
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:

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 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 WORDS ABOUT RESOURCES

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/.

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 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 res/drawable-hdpi/, res/drawable-xhdpi/, res/drawable-mdpi/, and res/drawable-ldpi/ directories, containing stock launcher icons (ic_launcher.png) for some of those densities (along with perhaps other icons).

Our AndroidManifest.xml file then references our ic_launcher icon:

```xml
<?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">

    <uses-sdk
        android:minSdkVersion="9"
        android:targetSdkVersion="15"/>
</manifest>
```
<supports-screens>
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"
    android:xlargeScreens="true"/>

<application>
    android:allowBackup="true"
    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>

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 (e.g., an extra-high-density device with our stub project), Android will take the next-closest one and scale it. So, for our stub project, Android would take the -hdpi icon and scale it up to work on an -xhdpi display, such as that found on the Samsung Galaxy Nexus.

**Drawable Resources and Eclipse**

Eclipse does not ship 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 Eclipse. Double-clicking on an image in the Package Explorer in Eclipse should bring up your default editor for that file type.

### Using Android System Drawables

Android has many icons and other images that are considered to be part of the SDK, in addition to many drawables that ship with the OS but are not considered to be part of the SDK. You are welcome to use these if you wish, though there are some things that you will wish to consider.

#### Directly Referencing SDK Drawables

Just as we can reference an @drawable/ic_launcher resource from our own project in places like the manifest (and, later on, from ImageView and ImageButton widgets), we can reference an Android system drawable. Instead of @drawable/, though, we use @android:drawable/, indicating that the icon in question comes from the SDK, not from our project.

You can find a list of drawables that are part of the Android SDK in, of all places, the JavaDocs for a strange little android.R.drawable class.

So, for example, ic_menu_share is listed as a constant on that class, and so we can reference @android:drawable/ic_menu_share anywhere that we would want to use a drawable resource.

However, there is a risk: device manufacturers are welcome to replace these drawables with their own artwork. That is not directly a problem, but if you are using some of your own icons in addition to icons that come from the SDK, you could get in trouble. Even though your icons might match those you see from the SDK in the emulator, or on some devices, it is entirely possible that on other devices, the SDK-supplied icons will look different than your custom ones. Your custom ones might be grayscale, while the device’s icons are in color, for example.

Hence, you should only directly reference SDK drawables this way in situations where, even if the device’s drawable is slightly different than you expect, your app will still look OK.
Copying Android System Drawables

You can find the actual artwork for these SDK drawables, and many others, in your SDK installation itself:

- Go to the directory where you installed the Android SDK
- In there, go into platforms/
- In there, choose an API level (as icons may look different in Android from OS version to version)
- In there, go into data/res/
- In there, look at the various drawable- directories, such as drawable-hdpi/

These icons are licensed under the Apache License 2.0, the same license that is used for the rest of Android, and therefore you are welcome to copy them for your own project, under the terms of that license.

This is a safer approach than directly referencing these system resources, because by copying them into your project and using your own local copies (with normal @drawable/ references), you are insulated from any changes that might be made by device manufacturers. On the other hand, it does require a bit more work, and it will make your app a tiny bit larger.

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:

1. px means hardware pixels, whose size will vary by device, since not all devices have the same “screen density” (the ~4” Galaxy Nexus and the ~10” Motorola XOOM have almost the same number of pixels in vastly different sizes)
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 device-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 text zoom levels, increasing and decreasing as needed based upon the user's chosen text zoom 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.

**Dimension Resources and 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:

![Figure 47: Eclipse, Editing Existing Dimension Resources](image)

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 your resources:

![Figure 48: Eclipse, Dimension Resources in XML Editor](image)

**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.
Tutorial #4 - Adjusting Our Resources

Our EmPubLite project has some initial resources, put there by the Android build tools when we created the project. 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.

Step #1: Changing the Name

Our application shows up everywhere as “EmPubLite” or “Blank Project”, depending upon how you created the project:

- 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.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).
In the Package Explorer, open up the `res/values/` folder — you should see a `strings.xml` file in there:

![Eclipse Package Explorer, Showing EmPubLite](image)

Double-click on `strings.xml` to open it in the string resources editor:

![Eclipse String Resources Editor](image)

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:

![Eclipse String Resources Editor with Details](image)

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).

**Outside of Eclipse**

Open up res/values/strings.xml in your favorite editor. You will find an element that looks like:

```xml
<string name="app_name">EmPubLite</string>
```

Change the text node in this element to EmPub Lite. Repeat the process for the title_activity_em_pub_lite resource, if there is one (depending on your tools version and such, there may not be one). Then save your changes, giving you something like:

```xml
<resources>
  <string name="app_name">EmPub Lite</string>
  <string name="hello_world">Hello world!</string>
  <string name="menu_settings">Settings</string>
</resources>
```

**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:
First, download the original image and save it somewhere on your development machine.

Then, follow the instructions for Eclipse or non-Eclipse users below.

**Eclipse**

From the Eclipse main menu, choose File > New > Other > Android. In the resulting dialog, choose “Android Icon Set” and press Next.

![Create Asset Set]

*Figure 53: 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.

![Fig 5.4: Eclipse Icon Set Wizard, Second Page](image)

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:
Click the “Browse…” button and open the molecule.png file you downloaded above. That will display the results in the wizard:
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.

If you run the resulting app, you will see that it shows up with the new name and icon, such as in the launcher:
Outside of Eclipse

We can use the Android Asset Studio to create launcher icons out of this image, if you have the Chrome browser.

Visit the Android Asset Studio Web site in Chrome. Then, click the “Launcher icons” link in the “Icon generators” portion of the home page.
Click on the “Image” button in the “Foreground” row. This will bring up a “file open” dialog — find and open the molecule.png file you downloaded previously. Automatically, the Studio will generate the icons we need:
Click the “Download .ZIP” button to download a ZIP archive file containing all the generated icons.

If you are having difficulty using the Android Asset Studio, you can [download the icons](https://commonsware.com) directly.

If you examine that ZIP file, you will see that it contains a `res/` directory with a series of `drawable` subdirectories, each containing a copy of `ic_launcher.png` for a given screen density. The ZIP file also contains a high-resolution image that we might use if we planned on uploading this app to Google Play, but we will not need that for the tutorials.

Copy the four `ic_launcher.png` files from the ZIP archive’s directories into the corresponding directories in your project. You may have to copy the whole `drawable-xhdpi/` directory, as that may not already exist in your project. If you are using Eclipse, you can drag-and-drop into the Package Explorer directly. If you prefer, you can drag-and-drop into the project as found on your development machine’s file system, but then you will need to press <F5> on your project in Eclipse to get it to reflect the changes you made behind Eclipse’s back.
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:

![EmPubLite with New Icons](image)

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.

In Our Next Episode…

... we will add a progress indicator to the UI of our tutorial project.
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:
THE THEORY OF WIDGETS

Ignoring the gray horizontal bars across the top of this screen, we see:

- an icon of a contact “Rolodex” card
- some text (“Phone-only (unsynced..)"
- a thin horizontal divider line
- another icon, showing a placeholder for a contact photo, in a frame
- two data entry fields
- an icon that looks like a downward-pointing arrowhead in a circle
- another thin horizontal divider line
- another piece of text (“Phone”)
- two more icons, that look like plus and minus signs in circles
- a button (“Home”)
- another data entry field
- two more buttons (“Done” and “Revert”) in some sort of bar across the bottom

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.

### 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
- 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 — you can see a thin scrollbar on the right side of the screen. The “Done” and “Revert” buttons, along with the scrollable container, are themselves anchored to sides of their parent container (e.g., the “Done”/“Revert” bar is anchored to the bottom).

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](https://commonsware.com). 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 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 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

An Android project’s src/ directory contains the 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 src/com/commonsware/android/). If you checked the checkbox in the Eclipse new-project wizard to create an activity — or if you used the command-line tools to create your project – you will have, in the innermost directory, a Java source file representing an activity class.

For the stub project we created earlier in this book, that sample class looks like this:

```java
package com.commonsware.empublite;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class EmPubLiteActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar
        return true;
    }
}
```
Dissecting the Activity

Let’s examine this Java code piece by piece:

```java
package com.commonsware.empublite;
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class EmPubLiteActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

The package declaration needs to be 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.

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.

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 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?

To explain that, we need to start thinking about layout resources and how resources are referenced from within Java code. We will get to that momentarily.

```java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar
    // if it is present.
    getMenuInflater().inflate(R.menu.activity_main, menu);
    return true;
}
```

The onCreateOptionsMenu() is used in Android to populate the action bar, or the options menu on older devices. We will discuss the action bar in an upcoming chapter. For now, just ignore this method.

Now, back to this mysterious R...

**Using XML-Based Layouts**

As noted earlier, 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 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).

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 the res/layout/main.xml file that came with our stub project:

```xml
<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">

    <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.,
**The Android User Interface**

/tools:context/ are there solely to support the Android build tools, Eclipse in particular, and do not affect the runtime execution of your project.

Android’s SDK ships with a tool (aapt) which uses the layouts. This tool should be automatically invoked by your Android tool chain (e.g., Eclipse, Ant’s build.xml). Of particular importance to you as a developer is that aapt generates an R.java source file within your project’s gen/ directory, 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](#).

**XML Layouts and Eclipse**

If you are using 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:

![Eclipse Graphical Layout Editor](image)

*Figure 62: Eclipse Graphical Layout Editor*

The “main.xml” sub-tab will show you the raw XML. The default “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.

Perhaps the biggest reason is to assist in the creation of tools for view definition, such as the aforementioned graphical layout editor in 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`).

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.

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.
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 125dp to indicate the widget should take up exactly a certain size (here, 125 density-independent pixels)
2. You can provide \texttt{wrap\_content}, which means the widget should take up as much room as its contents require (e.g., a \texttt{TextView} label widget's content is the text to be displayed).

3. You can provide \texttt{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 \texttt{fill\_parent}. This is an older synonym for \texttt{match\_parent}. \texttt{match\_parent} is the recommended value going forward, but \texttt{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 \texttt{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 \textit{their} requests are.

### Assigning Labels

The simplest widget is the label, referred to in Android as a \texttt{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 \texttt{TextView} instance. More commonly, though, you will create labels in XML layout files by adding a \texttt{TextView} element to the layout, with an \texttt{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 the XML instead (e.g., \texttt{@string/label}).

For example, in our last tutorial, we still are using the automatically-generated \texttt{res/layout/main.xml} file, containing, among other things, a \texttt{TextView}:

```xml
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    ...
</RelativeLayout>
```
BASIC WIDGETS

```xml
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <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>
```

Eclipse Graphical Layout Editor

The TextView widget is available in the “Form Widgets” portion of the Palette in the Eclipse graphical layout editor:

![Form Widgets Palette, TextView in Upper Left](image)

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:
Clicking on the resulting `TextView` in the Outline pane will set up the Properties pane with the various attributes of the widget, ready for you to change as needed:
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:

![String Resource Chooser](image)

*Figure 66: 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.

For example, in the `Basic/Label` sample project, you will find the following layout file:

```xml
<?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 67: The LabelDemo Sample Application](image-url)
**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:

```xml
<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:
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:
BASIC WIDGETS

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:

```java
public void someMethod(View theButton) {
    // do something useful here
}
```
Then, we could use this XML declaration for the Button itself, including android:onClick:

```xml
<Button
    android:onClick="someMethod"
...
/></Button>
```

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
<?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:
Eclipse Graphical Layout Editor

The ImageView widget can be found in the “Images & Media” portion of the Palette in the Graphical Layout editor:

![Image of Palette showing ImageView in the upper left]

*Figure 71: 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:

![Image of Eclipse Properties Pane](image.png)

*Figure 72: Scale Types in Eclipse Properties Pane*

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:
A choice of “centerCrop” will scale the image so that its shortest dimension fills the available space and crops the rest:
A choice of “fitXY” will scale the image to fill the space, ignoring the aspect ratio:
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
<?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"
    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:
**Eclipse Graphical Layout Editor**

The Graphical Layout’s Palette has a whole section dedicated primarily to `EditText` widgets, named “Text Fields”:
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 “Request Focus” item in the context menu (right-click over the EditText widget) allows you to indicate that this EditText should be the widget that receives the focus when this layout is loaded onto the screen. By default, the focus goes to the focusable widget that is first (i.e., closest to the upper-left corner), but you can override that using this attribute.

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:

![Figure 78: Text Fields InputType Dialog](image)

**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, 5px is 5 pixels, 10dp is 10 density-independent pixels, or 2mm is 2 millimeters.
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 5px for 5 pixels.

Colors

There are two types of color attributes in Android widgets. Some, like android:background, take a single color (or a graphic image 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 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.

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

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 selection.

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.
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.

![A Crash Dialog on Android 4.0.3](image-url)

*Figure 79: 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.
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 Eclipse

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.

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:

![Figure 80: Scaled Up Rendition of LogCat Icon](image)

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.
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.

There is a scrollbar towards the bottom of the main log area that will let you see more of your stack trace:
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.

The icon immediately to the right is the “clear” icon:
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 Eclipse, this is a matter of selecting the project, then choosing Project > Clean from the Eclipse main menu. Outside of Eclipse, `ant clean` accomplishes much the same thing.

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.

**Point Break**

If you are an experienced Eclipse user, you are welcome to use any of Eclipse's standard debugging capabilities with your Android app, such as breakpoints.
Whether you debug on an emulator or on a device (with “USB Debugging” enabled in Settings), your breakpoints and such should work normally.

Note, however, 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., StackOverflow), use the LogCat stack trace.
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. This works similarly to vbox and hbox in Flex and XUL, etc.

Flex and XUL 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 vis-à-vis 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.

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 weights...
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.

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
<?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:
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.
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.

**Eclipse Graphical Layout Editor**

The `LinearLayout` container can be found in the “Layouts” portion of the Palette of the Eclipse graphical layout editor:

![Figure 85: Layouts Palette in Eclipse Graphical Layout Editor](image)

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:
LINEAR LAYOUT AND THE BOX MODEL

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:

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:

- 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.
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
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
<?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:

```java
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)findViewByld(R.id.check);
        cb.setOnCheckedChangeListener(this);
    }

    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        if (isChecked) {
            cb.setText(R.string.checked);
        } else {
            cb.setText(R.stringunchecked);
        }
    }
}
```

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
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 88: The CheckBoxDemo sample application, with the checkbox unchecked](image)

![Figure 89: The same application, now with the checkbox checked](image)
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?

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:

![ToggleButtonDemo sample](image)

*Figure 90: The ToggleButtonDemo sample, showing an unchecked ToggleButton*
Otherwise, ToggleButton behaves much like CheckBox. You can put it in a layout file, as seen in the Basic/ToggleButton sample:

```xml
<?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.

**Eclipse Graphical Layout Editor**

Like CheckBox, the ToggleButton widget appears in the “Form Widgets” section of the Palette in the Graphical Layout editor. It looks like a button with the word “OFF” towards the top. You can drag it 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, 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
<?xml version="1.0" encoding="utf-8"?>
<RadioGroup
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   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/paper" />
</RadioGroup>
```
Using the stock Android-generated Java for the project and this layout, you get:

```
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>
```

![Figure 92: The RadioButtonDemo sample application](image)

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.

**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 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.

This is reminiscent of James Elliot’s RelativeLayout for use with Java/Swing.

**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).

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
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).

Order of Evaluation

It used to be that Android would use a single pass to process RelativeLayout-defined rules. That meant you could not reference a widget (e.g., via android:layout_above) until it had been declared in the XML. This made defining some layouts a bit complicated. Starting in Android 1.6, Android uses two passes to process the rules, so you can now safely have forward references to as-yet-undefined widgets.

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
<?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">
```
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:

![Figure 93: The RelativeLayoutDemo sample application](image)

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.
OTHER COMMON WIDGETS AND CONTAINERS

This will be clearer with an example. Here is a layout, from the Containers/RelativeOverlap sample, with a RelativeLayout holding two Button widgets:

```xml
<?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
        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:
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.

**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.

And, at this point, you can start getting frustrated. To paraphrase an old American candy commercial, drag-and-drop GUI building and `RelativeLayout` are two great tastes that do not taste great together.

The problem is that the complexity of the `RelativeLayout` rules makes it very difficult for the Graphical Layout editor to guess what you really mean when you drag a widget into the `RelativeLayout`. It will guess as best it can — for example, if you are dropping the widget near the edge of the `RelativeLayout`, it will assume you mean for the widget to be aligned with that edge. However, frequently, it will guess wrong, forcing you to modify the `RelativeLayout` XML directly via the other editor sub-tab or via the Properties pane to get the rules that you want.
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.

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:

```xml
<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.
**OTHER COMMON WIDGETS AND CONTAINERS**

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:

```xml
<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 automatically have their width set to match_parent, so they will fill the same space that the longest row does.

One pattern for this is to use a plain View as a divider (e.g., `<View android:layout_height = "2dip" android:background = "#0000FF" />` as a two-pixel-high blue bar across the width of the table).

**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.
OTHER COMMON WIDGETS AND CONTAINERS

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
<?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" />
    </TableRow>

    <View
        android:layout_height="2dp"
        android:background="#0000FF" />

    <TableRow>
        <Button
            android:id="@+id/cancel"
```
OTHER COMMON WIDGETS AND CONTAINERS

When compiled against the generated Java code and run on the emulator, we get:

![TableLayoutDemo sample application](image)

**Figure 95: The TableLayoutDemo sample application**

**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:

![Eclipse Layout Toolbar for TableLayout](image)

**Figure 96: 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.

Once you have started to populate the row and can select it, you will get some more toolbar buttons:

![Eclipse Layout Toolbar for TableLayout, with Row Selected](image)

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 [Containers/Scroll demo](https://commonsware.com/Android/2019/source/Containers/ScrollViewEx.xml)):

```xml
<?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"/>
    </TableLayout>
</ScrollView>
```
### Other Common Widgets and Containers

```xml
<TableRow>
    <View
        android:layout_height="80dp"
        android:background="#440000" />
    <TextView
        android:text="#440000"
        android:paddingLeft="4dp"
        android:layout_gravity="center_vertical" />
</TableRow>
<TableRow>
    <View
        android:layout_height="80dp"
        android:background="#884400" />
    <TextView
        android:text="#884400"
        android:paddingLeft="4dp"
        android:layout_gravity="center_vertical" />
</TableRow>
<TableRow>
    <View
        android:layout_height="80dp"
        android:background="#aa8844" />
    <TextView
        android:text="#aa8844"
        android:paddingLeft="4dp"
        android:layout_gravity="center_vertical" />
</TableRow>
<TableRow>
    <View
        android:layout_height="80dp"
        android:background="#ffaa88" />
    <TextView
        android:text="#ffaa88"
        android:paddingLeft="4dp"
        android:layout_gravity="center_vertical" />
</TableRow>
<TableRow>
    <View
        android:layout_height="80dp"
        android:background="#ffffff" />
    <TextView
        android:text="#ffffff"
        android:paddingLeft="4dp"
        android:layout_gravity="center_vertical" />
</TableRow>
</TableLayout>
</ScrollView>
```

164
Without the ScrollView, the table would take up at least 560 pixels (7 rows at 80 pixels each, based on the View declarations). There may be some devices with screens capable of showing that much information, but many will be 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:

![ScrollViewDemo sample application](image)

Notice how only five rows and part of the sixth are visible. By pressing the up/down buttons on the directional pad, 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 1.5 introduced 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.
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. A ScrollView or HorizontalScrollView can only have one child — if you want more than one, wrap the children in a suitable LinearLayout and put that inside the ScrollView or HorizontalScrollView.

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 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.
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.

Step #1: Renaming the Activity Class

If you created your project using the Eclipse new-project wizard or the command-line tools, you can skip to the next step in the tutorial.

Otherwise, you will find that your activity's class name is MainActivity. While that is a fine default name for an activity, the rest of the tutorials’ steps assume that the activity name is EmPubLiteActivity, and so we should rename this class.

To do this, right-click over MainActivity.java in the Eclipse Package Explorer and choose Refactor > Rename from the context menu. In the “New name” field, replace MainActivity with EmPubLiteActivity, then click Finish.

This not only renames the Java class file, but if you peek at your manifest, you will see that it also fixed the <activity> element, so the android:name attribute also contains EmPubLiteActivity.
Step #2: 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.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

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 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 99: EmPubLiteActivity, in Eclipse](image)
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, and save the resulting file.

Outside of Eclipse

Open res/layout/main.xml in your favorite text editor. If there is no such file, but you have another layout resource (e.g., activity_main.xml), rename it to main.xml. In res/layout/main.xml, find and delete the <TextView> element, then save the file. The resulting XML should look like:

```xml
<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">
</RelativeLayout>
```

Also, we no longer need the hello_world string resource. To remove it, open the res/values/strings.xml file in your favorite text editor. Find the <string> element that has a name of hello_world, delete that element, and save the file. The resulting XML should look like:

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">EmPub Lite</string>
    <string name="menu_settings">Settings</string>
</resources>
```

Step #3: Adding a ProgressBar

Now that the TextView is out of the way, we can add our ProgressBar in its place.
If you wish to make this change using Eclipse's wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**Eclipse**

Go back to res/layout/main.xml in Eclipse. In the “Form Widgets” portion of the tool palette, you will see three `ProgressBar` widget representations, in the form of circles:

![Palette](image)

*Figure 100: The ProgressBar Widget in the Tool Palette*

Drag the largest 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 101: The Centering Toolbar Buttons (Third and Fourth from Right)](image)

Then, you can save your file.

**Outside of Eclipse**

Go back to `res/layout/main.xml` in your favorite text editor. Delete the `<TextView>` element that was there. Replace it with a `<ProgressBar>` element as a child of the `<RelativeLayout>`, as shown below:

```xml
<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"/>

</RelativeLayout>
```

Then, you can save your file.

**Step #4: 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:
In Our Next Episode…

... we will attach a third-party library to our tutorial project.
If you are using Eclipse, 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 Eclipse, 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.

Clicking on a container 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 LinearLayout sample from a previous chapter – the entire LinearLayout is filled with the three Button widgets. In these cases, click on the widget in the Outline 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:

```xml
<include layout="@layout/thing_we_are_reusing" />
```

You can even assign the `<include>` element a width or height if needed, as if it were just a widget or container.

Eclipse 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. Just select the widget(s) 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:

![Extract as Include Dialog](image)

*Figure 103: Extract as Include Dialog*

By default, the tools 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, Eclipse will automatically wrap them in a `<merge>` element:

```xml
<?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.

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).

Similarly, if a widget is wrapped in a container, where the container is no longer necessary, “Remove Container” will get rid of the container.

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, right-click over the widget in Eclipse (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:
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 tab, you will find a series of drop-downs that allow you to tailor what the preview looks like:

Eclipse 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.
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.

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

In the abstract, adapters provide a common interface to multiple disparate APIs. More specifically, in Android’s case, adapters provide a common interface to the data model behind a selection-style widget, such as a listbox. This use of Java interfaces is fairly common (e.g., Java/Swing’s model adapters for JTable), and Java is far from the only environment offering this sort of abstraction (e.g., Flex’s XML data-binding framework accepts XML inlined as static data or retrieved from the Internet).
Android’s adapters are responsible for providing the roster of data for a selection widget plus converting individual elements of data into specific views to be displayed inside the selection widget. The latter facet of the adapter system may sound a little odd, but in reality it is not that different from other GUI toolkits’ ways of overriding default display behavior. For example, in Java/Swing, if you want a JList-backed listbox to actually be a checklist (where individual rows are a checkbox plus label, and clicks adjust the state of the checkbox), you inevitably wind up calling setCellRenderer() to supply your own ListCellRenderer, which in turn converts strings for the list into JCheckBox-plus-JLabel composite widgets.

**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:

```java
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.
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
<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:

```java
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;
```
With `ListActivity`, you can set the list adapter via `setListAdapter()` — in this case, providing an `ArrayAdapter` wrapping an array of nonsense 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?

---

---
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: big font, lots of padding, white text.

### 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, Google TV devices are not touchscreens at present. And some 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.

Selection 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 selection, or possibly multiple selections. 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, 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 will need to 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
<?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"
/>
```

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 nonsense words, but uses android.R.layout.simple_list_item_multiple_choice as the row layout:

```java
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", "auge", "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:
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.
Spin Control

In Android, the Spinner is the equivalent of the drop-down selector you might find in other toolkits (e.g., JComboBox in Java/Swing). Pressing the center button on the D-pad pops up a selection dialog for the user to choose an item from. You basically get the ability to select from a list without taking up all the screen space of a ListView, at the cost of an extra click or screen tap 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().

If you want 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
<?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:

```java
public class SpinnerDemo extends Activity
    implements AdapterView.OnItemSelectedListener {
    private TextView selection;
    private static final String[] items="lorem", "ipsum", "dolor",
```
Here, we attach the activity itself as the selection listener
(spinn.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:
Figure 109: The SpinnerDemo sample application, as initially launched

Figure 110: The same application, with the spinner drop-down list displayed
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 `setOnItemSelectedListener()` to register a selection listener, etc.

For example, here is an XML layout from the Selection/Grid sample project, showing a GridView configuration:

```xml
<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
        android:id="@+id/grid"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>
</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 = "5dp") and columns width (android:columnWidth = "100dp"), with the columns absorbing any “slop” width left over (android:stretchMode = "columnWidth").

The Java code to configure the GridView is:

```java
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;

    @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);
    }
```
@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
<?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 11: The GridDemo sample application, as initially launched](image-url)
Figur 112: The same application, 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.
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
<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:

```java
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", "morch", "vel",
       "ligula", "vitae", "arcu", "aliquet", "mollis",
       "etiam", "vel", "erat", "placerat", "ante"};
```
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:
Figure 113: The AutoCompleteDemo sample application, as initially launched
Figure 11.4: The same application, after a few matching letters were entered, showing the auto-complete drop-down.
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.
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 nonsense words used in earlier samples. However, this time, we want to have a more elaborate row, taking into account the length of the nonsense 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
<?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
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 nonsense words. The real fun comes when we override getView():
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:
ADAPTERVIEWS AND ADAPTERS

- 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 116: The Dynamic Sample Application](image)

*Figure 116: The Dynamic Sample Application*
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 calling `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](https://commonsware.com), a revised version of the Selection/Dynamic sample from before):

```java
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;
}
```
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:

```java
@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
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):

```java
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:

![Sound Settings Screen](image)

*Figure 117: Sound Settings Screen*
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.
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.
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. WebKit is an open source Web rendering engine that forms the heart of major Web browsers, such as Chrome and Safari. While the version of WebKit 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, 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 in Android is a mobile-optimized version, which introduces some differences compared to its desktop brethren
- WebKit, 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 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
<?xml version="1.0" encoding="utf-8"?>
    android:id="@+id/webkit"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```
The WebView Widget

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 drag a WebView out of the “Composite” section of the Eclipse tool palette and into a layout XML resource in the Graphical Layout editor:

![Figure 118: WebView in Eclipse Tool Palette](image)

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:

```java
package com.commonsware.android.browser1;
```
THE WebView Widget

```java
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.

The resulting activity looks like a Web browser, just with hidden scrollbars:
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.

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**

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:

```java
browser.loadData("<html><body>Hello, world!</body></html>", "text/html", "UTF-8");
```

You get:
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.
THE WEBVIEW WIDGET

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:

```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();
    }
}
```
Here, we load a simple Web page into the browser (loadTime()) that consists of the current time, made into a hyperlink to the /clock 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:
Selecting the link and clicking the D-pad center button will “click” the link, causing 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 has been implemented using a version of WebKit independent from any other browser efforts by Google, such as Chrome.
THE WebView Widget

As of Android 4.4, WebView is now based on Google’s open source Chromium project. That, in turn, is still based on WebKit.

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.
Defining and Using Styles

If you have done development using modern-day HTML, you will be familiar with Cascading Style Sheets (CSS). These provide two capabilities:

1. They let you define common characteristics of HTML elements in one place, applying them wherever as needed, to reduce repetition and simplify maintenance; and
2. They allow you to configure things about the HTML elements that pure HTML alone does not support

Android has similar constructs — styles and themes — for achieving similar ends. Styles and themes are another type of resource, akin to the layouts and strings and such that we have seen so far. Hence, the syntax of styles and themes is XML, rather than in CSS notation. However, the concepts and how they are employed are much like what you see with CSS.

This chapter will briefly explore the concept of styles, how you can create them, and how you can apply them to your own widgets.

**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](https://commonsware.com). This is a trivial project, with a full-screen
DEFINING AND USING STYLES

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 is the same as it was, with the addition of a style attribute:

```xml
<?xml version="1.0" encoding="utf-8"?>
<Button
    xmlns:android="http://schemas.android.com/apk/res/android"
    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
<?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).

There are no changes needed elsewhere in the project — nothing needs to be adjusted in the manifest, in the Java code of the activity, etc. Just defining the style and applying it to the widget gives us results:
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
<?xml version="1.0" encoding="utf-8"?
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```
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
<?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>
```
DEFINING AND USING STYLES

(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.

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
<?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.

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.Dialog or @android:style/Theme.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.NoTitleBar.Fullscreen:

```
<!-- Variant of the default (dark) theme that has no title bar and fills the entire screen -->
<style name="Theme.NoTitleBar.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.
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 Dalvik 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 Dalvik.
The Dalvik VM

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.

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 \texttt{javac} compiler. Then, however, the Java VM bytecodes created by \texttt{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.

The Easy Part

You have two choices for integrating third-party Java code into your project: use source code, or use pre-packaged JARs.

If you choose to use their source code, all you need to do is copy it into your own source tree (under \texttt{src/} in your project), so it can sit alongside your existing code, then let the compiler perform its magic.

If you choose to use an existing JAR, perhaps one for which you do not have the source code, place the JAR in the \texttt{libs/} directory in your Android project.

And that’s it, at least for Eclipse and Ant. 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.
JARs and Library Projects

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:

1. Right-click on the project in Eclipse’s Package Explorer and choose Build Path > Configure Build Path from the context menu
2. Switch to the “Order and Export” tab
3. Check the “Android Private Libraries” entry in the list, if it is not already checked

Hence, adding third-party code to your Android application is usually fairly easy.

Getting a library to actually work may be somewhat more complicated, however.

The Outer Limits

Not all available Java code will work well with Android. There are a number of factors to consider, including:

1. 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?
2. 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.
3. 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.
4. Interface: Does the Java code assume a console interface? Or is it a pure API that you can wrap your own interface around?
5. Operating System: Does the Java code assume the existence of certain console programs? Does the Java code assume it can use a Windows DLL?
6. 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 the new Java 7 release and has Java 7 bytecodes rather than Java 6?
7. **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.

**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. The original occurrence of this pattern is the “paid/free” application pair: two applications, one offered for free, one with richer functionality that requires a payment. Via a library project, the common portions of those two applications can be consolidated, even if those “common portions” include things like resources. 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 as well as Java code. If all you are looking to distribute is Java code, a JAR works just as well as an Android library project. But if you need to distribute layouts, themes, and the like, an Android library project is the solution.

**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. It supports third-party JAR files (e.g., /libs/).

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.
JARs and Library Projects

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 123: Android Library Project Properties, Library Section](image)

To create a library project for use with Ant, you can use the `android create lib-project` command. This has the net effect of putting an `android.library=true` entry in your project’s `project.properties` file.

Using a Library Project

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.

To do this in Eclipse, 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 124: Android Library Project Consumer Properties, Library Section](image)
For developing using Ant, you can use `android update project` command with the `--library` switch. This adds an entry like `android.library.reference.1=...` to your project’s `project.properties` file, where `...` is the relative path to your library project. You can add several such libraries, controlling their ordering via the numeric suffix at the end of each property name (e.g., `1` in the previous example).

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 Eclipse or `project.properties`.

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).

**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.
- While you can define entries in the manifest file for a library, at present, they are not used.
- 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 — sometimes, this is set to 1.5, which complains about such annotations.
The Android Support Package

The Android Support package is distributed by Google, containing classes (in JARs and an Android library project) 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.

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 they 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 standalone or in the form of “Android Support package” or “Android Support library”.

Getting It

You will find the Android Support package in your SDK Manager, in the “Extras” category towards the bottom of the tree:
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/`.

**Attaching It To Your Project**

From Eclipse, you can add the Android Support package to a project by right-clicking over the project and choosing Android Tools > Add Support library from the context menu.

Outside of Eclipse, you will want to find the `android-support-v4.jar` file installed in your `extras/` directory tree and add a copy to your project’s `libs/` directory. There is also an `android-support-v13.jar` and an Android library project associated with the Android Support package. However, unless specifically mentioned otherwise, this book will be referring to `android-support-v4.jar` when it refers to the Android Support package.
JAR Dependency Management

Suppose we have Project A that depends on Library B and Library C, where the B and C are Android library projects. Further suppose that Project A, Library B, and Library C all need the Android Support package, so their projects are set up with access to it (e.g., having android-support-v4.jar in libs/).

You might think that we would somehow wind up with three copies of this support JAR in our APK. Fortunately, that is not the case. Android recognizes, based on filename, that these are the same JAR and therefore will only include one.

However, what happens if Google releases an update to the Android Support package, and you download the update?

Initially, nothing happens, if the support JARs are copied into your projects. If, however, you copy a fresh JAR into, say, Library C, without updating Library B or Project A, you will get a build error. Android will detect that while all three projects refer to the same JAR by name, the JARs themselves are different (based on SHA1 hash), and the build will fail. You will need to ensure that all three projects get the updated JAR.

The general rule of thumb is:

- Every Android library project needing the JAR should have the latest JAR, either in its own libs/ directory or because it depends upon another Android library project needing the JAR
- An Android app that depends upon an Android library project that can supply the JAR should not have its own copy of that JAR in its own libs/ directory
We will want to use a library named ActionBarSherlock in our project. This Android library project gives us a backwards-compatible edition of a UI construct known as the action bar, which we will examine in greater detail in the next chapter. So, in this tutorial, we will download and set up ActionBarSherlock.

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.

**Step #1: Downloading and Unpacking ActionBarSherlock**

Visit the ActionBarSherlock site and download the ZIP file (or tarball, if you prefer) from the home page for the current ActionBarSherlock release (4.3.1 at the time of this writing).

For the purposes of this tutorial, Eclipse users should take the actionbarsherlock/ directory out of the ZIP file and place it on your desktop. Non-Eclipse users should take the actionbarsherlock/ directory out of the ZIP file and place it in a directory parallel to your EmPubLite/ directory.

While the ZIP file will contain other directories, such as actionbarsherlock-i18n/, you do not need them for these tutorials.

Note that a copy of a compatible version of ActionBarSherlock can be found in the book’s GitHub repository in its proper place relative to the EmPubLite projects in
Step #2: Adding the Library to Your Project

Of course, merely downloading ActionBarSherlock does not somehow magically make it available to us. We need to add it to the EmPubLite project if we want to take advantage of its capabilities.

If you wish to make this change using Eclipse's wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

Eclipse

First, we need to create a second Eclipse project, this one to hold ActionBarSherlock. Since ActionBarSherlock does not ship with Eclipse project files, we will have to load it from source.

To do that:

- Choose File > New > Project... from the Eclipse main menu
- Choose “Android Project from Existing Code” from the list of project types and click “Next >”
- Click the “Browse...” button next to the “Root Directory” field, browse to the actionbarsherlock directory you created above, then click OK
- Check the “Copy projects into workspace” checkbox
- Click “Finish” to create the project

You may see a red error message in the console about “not able to find android-14”. This is because ActionBarSherlock ships with API Level 14 as the build target; if you
do not have this, you will get this message. However, the Android build tools in 
Eclipse will automatically fix up your project to use another available build target, 
and so usually there is nothing that you need to do to address this error message.

If you see some red “X” error indicators over the src/ and res/ folders, right-click 
over the project and choose Properties from the context menu. In the Properties 
window, choose Android, then set the build target to API Level 14 or higher. Click 
“OK” to close up the Properties window. Then, from the Eclipse main menu, choose 
Project > Clean, ensure the ActionBarSherlock project is checked in the list of 
projects, and click “OK”. This should eliminate the error indicators.

If you are still getting errors, and an examination of the ActionBarSherlock code 
indicates that the complaints are about @Override annotations on methods that are 
implementing an interface, rather than truly overriding a superclass method, you 
need to adjust your Eclipse compiler compliance level to be 1.6, instead of 1.5. Even if 
you already did this at the workspace level, you may need to do it at a project level. 
To do this:

1. Right click over the project name and choose Properties from the context 
   menu
2. Click on “Java Compiler” in the tree on the left
3. Choose 1.6 from the “Compiler compliance level” drop-down
4. Click “Apply”, then “OK”

Note that if you use the [copy of ActionBarSherlock in this book’s GitHub repository](https://commonsware.com), 
then you can skip the above steps and just import the project directly into Eclipse 
(e.g., File > Import from the main menu).

To add the project as a library on EmPubLite, right-click over the EmPubLite project 
and choose Properties from the context menu. In the Properties window, choose 
Android, then click “Add...” in the Library group box, towards the bottom, on the 
right. In the list of library projects that appears, choose ActionBarSherlock, then 
click “OK”. The Library group box should then resemble the following:
Click “OK” to close up the Properties window.

Then, right-click over the EmPubLite project and choose Build Path > Configure Build Path from the context menu. In the Order and Export tab, ensure that “Android Private Libraries” appears and is checked. If this item does not appear at all, you may be on an earlier version of the ADT plugin, and you should be OK.
If you see error messages in the Eclipse console, complaining about JARs with the same name and different contents, please read the note at the end of Step #1 of this tutorial.

### Outside of Eclipse

Switch to the ActionBarSherlock project directory and run:

```
android update lib-project --path .
```

This will create the `build.xml` and other necessary files for command-line builds with ActionBarSherlock.

If you get an error message about missing a build target, run the `android list targets` command to list all of the available build targets on your development machine. Then, run:

```
android update lib-project --path . --target ...
```

where the `...` is replaced by the name of your newest build target (e.g., `android-14`). This is necessary if ActionBarSherlock is shipped with a build target that is
TUTORIAL #6 - ADDING A LIBRARY

unavailable on your development machine. Any build target for API Level 14 or higher should work fine.

Then, switch to the EmPubLite project directory and run:

    android update project --path . --library ../actionbarsherlock

This tells Android to update your project.properties file to contain something resembling the following:

    target=android-15
    android.library.reference.1=../actionbarsherlock

In Our Next Episode…

... we will configure the action bar on our tutorial project
Options Menus and the Action Bar

Like applications for the desktop and some mobile operating systems, Android supports activities with “application” menus. Some Android devices will have a dedicated MENU key for popping up the menu; other devices will offer alternate means for triggering the menu to appear, such as an on-screen soft button.

However, the preferred approach nowadays is to have your menu choices be part of what Android calls the action bar. The action bar is a strip across the top of your activity that provides users with ways of performing actions within that activity, such as toolbar buttons. While the action bar is only native to Android in Android 3.0 and higher, there are ways to get an action bar in Android 2.x devices as well, through an Android library project known as ActionBarSherlock.

Bar Hopping (a.k.a., Terminology)

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.
OPTIONS MENUS AND THE ACTION BAR

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.

![Status Bar and Title Bar](image)

*Figure 128: 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.).
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”:

*Figure 129: Action Bar and System Bar*
OPTIONS MENUS AND THE ACTION BAR

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+, 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
Android 4.2, 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.
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:
This is why you will see references to “options menu” scattered throughout the Android SDK and in (:cough:) older Android books.

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**

You have two ways of getting an action bar into your apps. In the long term, you will be able to simply use Android’s native implementation. In the short term, however, most likely you will want to use ActionBarSherlock.
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 app will only ever run on Android 3.0 or higher devices, using the native action bar is a fine choice. However, at the time of this writing, relatively few devices run Android 3.0 and higher. You can find out how many devices are running various versions of Android via the “Platform Versions” portion of the “Device Dashboard” section of the Android Developers Web site. This is updated monthly and shows who is using what, in the form of a table and a pie chart:

![Platform Versions Chart from December 2013 (image courtesy of Google)](image)

Until a preponderance of devices runs Android 3.0 or higher, you would be stuck with the legacy options menus on older devices, and that would be sad.

**ActionBarSherlock**

You might think that the Android Support package, with its focus on backports, would have some facility for adding an action bar to apps running on older devices. Alas, it did not, until August 2013, meaning that there was a ~2.5 year gap where older devices had no official backport.
Various third-party projects implemented action bars to try to fill this gap, and none has done nearly as well as has ActionBarSherlock.  

ActionBarSherlock, in effect, extends the Android Support package, adding a backported action bar for apps running on devices prior to API Level 14 (Ice Cream Sandwich). 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).

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.

Installation

You 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. We will go through all the steps of this process in an upcoming tutorial.

Base Activity Class

You will need to adjust your project 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.

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:
NOTE: If you use this sample app, or any other one that uses ActionBarSherlock, you will need to update its configuration to point to your own copy of ActionBarSherlock’s Android library project.

What We Will Be Doing

In this book, we will generally be using ActionBarSherlock. Right now, most developers should still be targeting Android 2.x devices, and that will remain the case well into 2013. By late 2013, Android 2.x may have a small enough user base that you could consider dropping ActionBarSherlock… assuming nothing new shows up that ActionBarSherlock fixes.

For apps that are only targeting API Level 11 or higher, you can elect to skip ActionBarSherlock and use the pure native action bar implementation. A few examples in this book — mostly ones that for other reasons only work on API Level 11+ — will go that route.
Setting the Target

Whether you are using ActionBarSherlock or not, you will want to arrange to target API Level 11 or higher at runtime. That involves setting the android:targetSdkVersion attribute of the <uses-sdk> element of your manifest.

We see this in the same ActionBar/ActionBarDemo manifest originally shown above:

```xml
<?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>
```

Specifically, we have android:targetSdkVersion set to 14. 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 “…” affordance 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 seems to resolve this.

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.

**Minding Narrow**

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,
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.
In Eclipse’s manifest editor, this appears as the “UI options” field on the Application
tab or in the details for a specific selected activity.

**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:

![Android Menu]

### Figure 135: Eclipse Menu Resource Editor

Or, you can work with the raw XML, such as res/menu/actions.xml from ActionBar/ActionBarDemo:

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/add"
    android:actionLayout="@layout/add"
    android:icon="@android:drawable/ic_menu_add"
    android:showAsAction="ifRoom"
    android:title="@string/add"/>
  <item
    android:id="@+id/reset"
    android:icon="@android:drawable/ic_menu_revert"
    android:showAsAction="always|withText"
    android:title="@string/reset"/>
  <item
    android:id="@+id/about"
    android:icon="@android:drawable/ic_menu_info_details"
```
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.

**Action Layouts**

What happens if you want something other than a button to appear in the toolbar? Suppose you want a field instead?

Fortunately, this is supported. Otherwise, this would be a completely pointless section of the book.

In addition to the menu item configuration options mentioned above, you can also specify android:actionLayout (the “Action layout” field in Eclipse). This will be a reference to a layout XML resource that you want to have inflated into the action bar.
instead of a toolbar button. Obviously, since the action bar is only so big, you will need to be judicious about your use of space, which is why the res/layout/add.xml resource, referred to from our “add” item, is just a LinearLayout holding onto a TextView label and an EditText field:

```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="wrap_content"
    android:orientation="horizontal">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Word:"
        android:textAppearance="@android:style/TextAppearance.Medium"/>
    <EditText
        android:id="@+id/title"
        android:layout_width="0px"
        android:layout_weight="1"
        android:layout_height="wrap_content"
        android:layout_marginLeft="4dip"
        android:layout_marginTop="4dip"
        android:imeActionId="1337"
        android:imeOptions="actionDone"
        android:inputType="text"
        android:width="100sp"/>
</LinearLayout>
```

Some notable features of our layout include:

1. We add an android:textAppearance attribute to the TextView representing our “Word:” caption. The android:textAppearance attribute allows us to define the font type, size, color, and weight (e.g., bold) in one shot. We specifically use a “magic value” of @android:style/TextAppearance.Medium, so the caption matches the styling of the “Reset” label on our other menu item we promoted to the action bar.
2. We specify android:width="100sp" for the EditText widget, to provide room for other contents within our split action bar.
3. We specify android:inputType="text" on the EditText widget, which, among other things, will restrict us to a single line of text.
4. We also specify android:imeActionId and android:imeOptions on the EditText widget to control the “action button” of the soft keyboard, so we get control when the user presses <Enter> on the soft keyboard. That’s a bit beyond the scope of what we have covered so far with EditText — if you
want to learn more about it, you will want to read the chapter on the input method framework.

So, given our menu resource XML listed earlier in this chapter, we are requesting:

- A custom action view (@layout/add), if there is room, and
- An action overflow item, named @id/reset

**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/ActionBarDemo sample project:

```java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);
    configureActionItem(menu);
    return (super.onCreateOptionsMenu(menu));
}
```

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. We will discuss that configureActionItem() method call shortly.

Note that the specific implementations of Menu and MenuInflater will depend upon whether you are using ActionBarSherlock or not — if you are, you will need to use the Sherlock versions (com.actionbarsherlock.view.Menu and com.actionbarsherlock.view.MenuInflater) instead of the standard Android SDK ones (android.view.Menu and android.view.MenuInflater).

**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:

```java
@Override
public boolean onOptionsItemSelected(MenuItem item) {
```
if (item.getItemId() == R.id.reset) {
    initAdapter();
    return(true);
}

return(super.onOptionsItemSelected(item));

You will be passed a MenuItem (either android.view.MenuItem or com.actionbarsherlock.view.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() or getSupportActionBar(), depending on whether you are using ActionBarSherlock), to enable this behavior.

**Attaching to Action Layouts**

This works nicely for our reset action overflow item. What about that other menu item, where we requested our custom action view layout?

That is where that configureActionItem() method comes into play, that we called from onCreateOptionsMenu():

```java
private void configureActionItem(Menu menu) {
    EditText add =
            (EditText)menu.findItem(R.id.add).getActionView().
            findViewById(R.id.title);

    add.setOnEditorActionListener(this);
}
```

Here, we ask the Menu to find the MenuItem object associated with our given item ID (@id/add). We then retrieve our inflated layout by a call to getActionView(). Finally, we get at the EditText widget by means of our old standby, findViewById(). Note that we have to call findViewById() on the inflated layout, not the activity.
Given this widget, we can now configure it as we see fit. In this case, we call `setOnEditorActionListener()`, indicating to Android that we want to get control when the user presses <Enter> or clicks the action button in the lower right corner of most soft keyboards. We will see what we do on that event shortly.

**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 nonsense words, and we want to display these in a list. However, in addition, we want to allow the user to add words to the list and revert the list to its original state.

`ActionBarDemoActivity` is a `SherlockListActivity` — an ActionBarSherlock equivalent of the `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:

```java
private void initAdapter() {
    words = new ArrayList<String>();
    for (String s : items) {
        words.add(s);
    }

    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 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 overflow menu, we call `initAdapter()` again, which gives our `ListActivity` a fresh set of nonsense words to display:
When the user presses <Enter> or clicks the “Done” button on the soft keyboard while typing in our EditText, control routes to our activity’s onEditorAction() method, which is required of a TextView.OnEditorActionListener, which itself is required because we are supplying the activity as the parameter to setOnEditorActionListener():

```java
@Override
public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {
    if (event == null || event.getAction() == KeyEvent.ACTION_UP) {
        adapter.add(v.getText().toString());
        v.setText("");

        InputMethodManager imm =
            (InputMethodManager) getSystemService(INPUT_METHOD_SERVICE);
        imm.hideSoftInputFromWindow(v.getWindowToken(), 0);
    }
    return(true);
}
```

We know the user has completed entering a word when onEditorAction() is invoked and the supplied KeyEvent is null or is ACTION_UP (meaning the user lifted their finger off of the key). At that point, we do three things:

1. We grab the nonsense word out of the field (supplied to us as a TextView parameter to onEditorAction()) and we add() it to our ArrayAdapter. The add() method appends this word to the end of the words in our list. This works because we used an ArrayList for the ArrayAdapter, and ArrayList objects’ contents can be modified at runtime (unlike static string arrays). A side effect of calling add() is that the ArrayAdapter will tell its attached ListView that the contents of the list changed, so the ListView will redraw itself and our new word appears at the bottom.

2. We clear out the field, so the user knows that we have accepted the new word.

3. We use the InputMethodManager to hide the soft keyboard, which will not automatically go away if the user presses <Enter>.
The net result of all of this is that we have an activity with our customized action bar:

![Figure 136: ActionBarDemo, As Initially Launched, on Android 4.3](image)

where the user can also type in a nonsense word into the field:
If the user presses `<Enter>` or clicks that “Done” button in the lower right corner of the soft keyboard, the nonsense word is added to the end of the list:
Figure 138: ActionBarDemo, With Additional Word, 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:
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:
Finally, on Android 2.x devices, pressing the MENU key will cause a classic options menu to appear:

*Figure 140: ActionBarDemo, on Android 4.0.3 Normal Screen, with Overflow*
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:
To accomplish this, you can use FEATURE_ACTION_BAR_OVERLAY, as is illustrated in the ActionBar/Overlay sample project.

This is nearly identical to the ActionBar/ActionBarDemo sample project, with just a few changes, mostly in the onCreate() method of our activity:

```java
@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);
    getSupportActionBar().setBackgroundDrawable(d);
    getSupportActionBar().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
<?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](https://commonsware.com). 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:
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.

**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.
Options Menus and the Action Bar

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.

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 “...” affordance 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 “...” affordance 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 “...” affordance and those that do not

Visit the Trails!

In addition to this chapter, you can learn more about navigation options in the action bar (e.g., tabs) and learn about action modes, which temporarily replace the action bar with new items for use with contextual operations.
Now that we have added ActionBarSherlock to our project, it is time to put it to use, adding 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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book’s GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.

Starting in this tutorial, we will now begin editing Java source files. Eclipse users should try to remember two useful shortcut key combinations:

- <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
- <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.

**Step #1: Setting the Theme and Splitting the Bar**

In order to use ActionBarSherlock, we need to apply a theme to our activities. As discussed previously, a theme applies a certain look and feel to the activities, such as color scheme. We need to use a theme from ActionBarSherlock itself for our action
bar to work. And, since we need the theme for all of our activities, we will set up the theme application-wide.

Also, over time, we may add enough items to our action bar that, on phones in portrait mode, things get too crowded. To combat this threat, we will also tell Android to split our action bar on narrow screens, giving us space at the top and bottom of the screen for our items.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**Eclipse**

Back in AndroidManifest.xml, click over to the Application sub-tab of the editor. Click the “Browse...” button to the right of the Theme field, choose “Theme.Sherlock.Light.DarkActionBar” from the list, then click OK.

Also, click the “Select...” button next to the “Ui options” field, check the checkbox next to “splitActionBarWhenNarrow”, and click “OK” to accept that change.

Your Application sub-tab’s “Application Attributes” area should now resemble:

![Figure 144: Eclipse Manifest Editor, Application Sub-Tab](image)

You can now save your changes (e.g., <Ctrl>-<S>).

Outside of Eclipse

Back in AndroidManifest.xml, add android:theme="@style/Theme.Sherlock.Light.DarkActionBar" and android:uiOptions="splitActionBarWhenNarrow" attributes to the <application> element, replacing any existing attributes with the same name. Your resulting manifest should resemble:

```xml
<?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">

    <uses-sdk
        android:minSdkVersion="9"
        android:targetSdkVersion="15"/>

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

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/Theme.Sherlock.Light.DarkActionBar"
        android:uiOptions="splitActionBarWhenNarrow">

        <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>
```

Step #2: Changing to SherlockFragmentActivity

The final step to simply have an action bar is to have our activity inherit from a suitable ActionBarSherlock base class. Ordinarily, we might choose SherlockActivity. However, in a future tutorial, we will start working with
fragments, and so with that in mind, we will set up EmPubLiteActivity to inherit from SherlockFragmentActivity.

If you open up EmPubLiteActivity, you will see that our current implementation is untouched from what Android code-generated for us when we created our project:

```java
package com.commonsware.empublite;
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class EmPubLiteActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar
        // if it is present.
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }
}
```

Simply change it from extends Activity to extends SherlockFragmentActivity. You will need to adjust your imports to import com.actionbarsherlock.app.SherlockFragmentActivity (Eclipse users can simply press <Ctrl>-<Shift>-<O> to automatically fix up the imports). Also, delete the onCreateOptionsMenu() implementation that was code-generated for you.

The result should resemble:

```java
package com.commonsware.empublite;
import android.os.Bundle;
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class EmPubLiteActivity extends SherlockFragmentActivity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```
If Eclipse starts complaining about your having an “inconsistent hierarchy” for EmPubLiteActivity, that is because your EmPubLite project has a different android-support-v4.jar file in it than does the ActionBarSherlock library project. Copy the android-support-v4.jar from your project’s libs/ directory into the libs/ directory of ActionBarSherlock, so both projects work off of the same JAR contents. You can learn more about this in the chapter on libraries.

**Step #3: Defining Some Options**

Of course, our current action bar is very boring.

Very, very boring.

To make it more useful and worthy of its screen space, we need to start adding some action items. Right now, we will add a couple of low-priority action items, for a help screen and an “about” screen.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**Eclipse**

**WARNING:** Bugs in the R22.6.1 version of the ADT plugin for Eclipse prevent the regular Eclipse-specific instructions from working. Make sure that you are on R22.6.2 or higher. You can check your version of the ADT plugin in Eclipse by opening up the About dialog (Help > About Eclipse Platform), then clicking on “Installation Details”, and looking at the Version column values for the “Android…” add-on entries. All should be 22.6.2 or higher.

Open the res/menu/ folder in your project.

In there, you may find a single file, whose name may vary (main.xml, activity_main.xml, em_pub_lite.xml, etc.). Whatever it is called, right-click over it, choose Refactor > Rename from the context menu, and rename it to options.xml. Then, double-click on this file to open it in an Eclipse resource editor.

If you do not find any file in res/menu/, then 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.

Click on the code-generated menu item (e.g., action_settings, menu_settings) and change the following values:

- In “Id”, enter @+id/help
- Delete the 100 from “Order in category”
- In “Icon”, enter @android:drawable/ic_menu_help

If there are no code-generated menu items, click the “Add” button to add one with the above values.

Note that there is an unpleasant bug, whereby copy-and-paste in structured editors like this one is broken, so you will have to type in the values by hand, or paste things in the XML directly.

Also, click the “Browse...” button to the right of the Title field. Click the “New String...” button towards the bottom of the dialog, to bring up the string resource editor:

![Create New Android String Window](image)

*Figure 145: The String Resource Editor*
Fill in Help in the String field and help in the “New R.string” field, then click “OK” to define this string resource. Choose the help string resource in the resource chooser, then click “OK” to use it. Save your file (e.g., <Ctrl>-<S>).

Next, we want to add a new menu item, so click the “Add...” button to the right of the list of menu options. Note that when you click the “Add...” button, you will initially be offered to create a child of the currently-selected item — click the “Create a new element at the top level, in Menu” radio button to be able to create a new item.

This time, use the following values:

- In “Id”, enter @+id/about
- In “Title”, create a new R.string.about string resource, with a value of About
- In “Icon”, enter @android:drawable/ic_menu_info_details
- In “Show as action”, click the “Select...” button and choose “never” from the list
- Save your changes (e.g., <Ctrl>-<S>)

Outside of Eclipse

Delete the existing file in res/menu/ and create a new res/menu/options.xml file, filling in the following XML content:

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/help"
        android:icon="@android:drawable/ic_menu_help"
        android:showAsAction="never"
        android:title="@string/help"/>

    <item
        android:id="@+id/about"
        android:icon="@android:drawable/ic_menu_info_details"
        android:showAsAction="never"
        android:title="@string/about"/>

</menu>
```

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
<?xml version="1.0" encoding="utf-8"?>
<resources>

</resources>
```
### Step #4: 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 a Sherlock-flavored version of `onCreateOptionsMenu()` and an `onOptionsItemSelected()` method to `EmPubLiteActivity`, as follows:

```java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    new MenuInflater(this).inflate(R.menu.options, menu);

    return (super.onCreateOptionsMenu(menu));
}

@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 this 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 (and `ActionBarSherlock` on Android 2.x) 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. We need to do a bit more work to enable that icon tap, but we will worry about that when we actually have something to do when the user taps on it.

To get this to compile, you will need to add some imports as well:

```java
import com.actionbarsherlock.view.Menu;
import com.actionbarsherlock.view.MenuInflater;
import com.actionbarsherlock.view.MenuItem;
```

(Eclipse users can just use `<Ctrl>-<Shift>-<O>` to import these, choosing the “Sherlock” versions of the classes when prompted)

Also, the pasted code may be poorly formatted. Eclipse users can press `<Ctrl>-<Shift>-<F>` to format the code into something reasonable.

**Step #5: Running the Result**

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. To display our options, you would need to press `MENU:`
On devices that lack a dedicated MENU button, the action bar will have a “…” icon somewhere on the split action bar:
Pressing that brings up a menu showing our items:
In Our Next Episode…

... we will define our first new activity on the tutorial project.
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 created a process. That process contains:

- A copy of the Dalvik VM, 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
BACK, HOME, and Your Process

Suppose, 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. 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).

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, 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.

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.
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.

This, of course, 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.

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
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
<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. Outside of Eclipse, you can just create a new Java source file, containing a public Java class that extends Activity directly or indirectly.

From Eclipse, you also have the option of using the new-class dialog, which you get by right-clicking over the Java package you want to contain this activity and choosing New > Class from the context menu.
Supply your class name (e.g., `OtherActivity`) and indicate its superclass (e.g., `com.actionbarsherlock.app.SherlockActivity`), then click “Finish” to add the empty class.

You can then add an `onCreate()` method to the activity, 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.

In Activities/Explicit, our second activity is `OtherActivity`, with pretty much the standard bare-bones implementation:

```java
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
<?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: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.

If you are using Eclipse, and you bring up the manifest in the editor, you can switch over to the Application sub-tab and look at the bottom half of the screen at the “Application Nodes” area:

![Figure 150: The Eclipse Manifest Editor Application Nodes](image)

Clicking the “Add...” button will allow you to choose to add “a new element at the top level, in Application” and add an activity:
Activities and Their Lifecycles

Clicking “OK” will give you a blank entry in the “Application Nodes” list, and you can fill in the details on the right. The only one that is essential is the “Name”, which will be the name of your activity — you can pick it out of a list via the “Browse...” button to the right of the “Name” field.
ACTIVITIES AND THEIR LIFECYCLES

You can also elect to supply a “Label”, pointing to a string resource which will populate the gray title bar of your activity. By default, you will inherit the label from the <application> element.

Outside of Eclipse, adding an activity to the manifest is a matter of adding another <activity> element to the <application> element:

```xml
<?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-sdmk
    android:minSdkVersion="7"
    android:targetSdmkVersion="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">
```
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:

```java
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 startActivityForResult() with an explicit Intent, identifying OtherActivity:

```java
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) {
```
Our launched activity shows the button:

![Figure 153: The Explicit Intents Demo, As Launched](image)

Clicking the button brings up the other activity:
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:
ACTIVITIES AND THEIR LIFECYCLES

- 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.

As it turns out, we can, as is illustrated in the Activities/Launch sample project.

We have a LaunchDemo activity that uses a layout containing two EditText widgets and a Button, among other things:

```xml
  <LinearLayout android:layout_width="match_parent" android:layout_height="wrap_content" android:orientation="horizontal">
    <TextView android:layout_width="wrap_content" android:layout_height="wrap_content" android:paddingLeft="2dp" android:paddingRight="4dp" android:text="@string/location"/>
    <EditText android:id="@+id/lat" android:layout_width="0dip" android:layout_height="wrap_content" android:layout_weight="1" android:inputType="numberDecimal|numberSigned" android:hint="@string/lat"/>
    <EditText android:id="@+id/lon" android:layout_width="match_parent" android:layout_height="wrap_content" android:inputType="numberDecimal|numberSigned" android:hint="@string/lon"/>
  </LinearLayout>
</LinearLayout>
```

---

Special Creative Commons BY-NC-SA 4.0 License Edition
The Button is tied to a showMe() method on the activity itself, where we want to bring up a map on the latitude and longitude entered into the EditText widgets:

```java
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 {
    private EditText lat;
    private EditText lon;

    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);

        lat=(EditText)findViewById(R.id.lat);
        lon=(EditText)findViewById(R.id.lon);
    }

    public void showMe(View v) {
        String _lat=lat.getText().toString();
        String _lon=lon.getText().toString();
        Uri uri=Uri.parse("geo:"+_lat+","+_lon+"?z=15");

        startActivity(new Intent(Intent.ACTION_VIEW, uri));
    }
}
```

Just as HTTP uses a verb and a URI, Android uses an action and a Uri. The standard Uri structure to express a location is one that uses the geo: scheme, followed by the
latitude and longitude in decimal degrees, followed by a zoom level from 1 to 21, with higher values representing tighter levels of zoom (e.g., geo:37.760829,-122.416111?z=15). Assembling this as a string is a matter of concatenation, but afterwards we need to convert it to a Uri via calling Uri.parse(). Then, we can use an action called ACTION_VIEW to try to display a map on that location.

When launched, the user is presented with our data entry form:

![Image](image.png)

Figure 155: The Launch Demo, As Initially Launched

We can fill in a latitude and longitude, replacing the values displayed as the “hints” (supplied by the android:hint attributes):
If the device has one application that responds to an `ACTION_VIEW` Intent on a `geo:` scheme, clicking the “Show Me!” button will bring up a map on that location:
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.

Note that previous versions of this book used a simpler `geo:latitude,longitude` Uri, but that no longer seems to be supported by the Google Maps application.

**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.
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:

```java
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:

```java
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) {
```

301
Activities and Their Lifecycles

```java
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.

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
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:
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`).
Conversely, anything that steals your user away from your activity — 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().

**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
Activities and Their Lifecycles

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:

```java
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);
    }
}
```
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:
If we click the button on the first activity to start up the second, we get:

If we press the BACK button on the second activity, returning to the first activity, we see:

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:

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.
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():

```java
Intent i = new Intent(this, HomeActivity.class);
i.setFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT);
startActivity(i);
```
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:

```java
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.
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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book’s GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.

**Step #1: Creating the Stub Activity Class**

First, we need to define the Java class for our new activity, SimpleContentActivity.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).
Eclipse

Right click over the `com.commonsware.empublite` package in the `src/` folder of your project, and choose New > Class from the context menu. That will bring up a dialog box for defining the new class:

![Image of Eclipse New Class dialog]

*Figure 158: Eclipse New Class Activity*

Fill in `SimpleContentActivity` in the “Name” field. Then, click the “Browse…” button next to the “Superclass” field, and type in `Sherlock` in the field at the top of the resulting dialog:
Choose SherlockFragmentActivity from the list, and click “OK” to close up that dialog. Then, click “Finish” to close up the new-class dialog. This will create your new Java class, albeit with no methods. That is OK, as we do not need any methods at this time.

**Outside of Eclipse**

Create a src/com/commonsware/empublite/SimpleContentActivity.java source file, with the following content:

```java
package com.commonsware.empublite;

import com.actionbarsherlock.app.SherlockFragmentActivity;

public class SimpleContentActivity extends SherlockFragmentActivity {
}
```

**Step #2: Adding the Activity to the Manifest**

If an activity was created in a forest and nobody was there to see it, does the activity really exist?
Tutorial #8 - Setting Up an Activity

Or, to be a bit less oblique, simply creating the activity class is insufficient for it to be used. We also need to add an <activity> element to the manifest, so other parts of our code can start up the activity.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

Eclipse

Double-click on AndroidManifest.xml in your project, and click over to the Application sub-tab. Scroll down to the “Application Nodes” list, then click the “Add...” button adjacent to that list. Choose “Activity” from the list of available items, and click “OK” to close up the dialog. This adds an empty activity entry in your manifest:

![Figure 160: Manifest Application Nodes, With New Activity]

Click the “Browse...” button to the right of the “Name” field. There will be a short pause while Eclipse scans your project for subclasses of Activity. In a moment, a list should appear, with SimpleContentActivity in it. Click on SimpleContentActivity, then click the “OK” button to make this choice. At this point, you can save your file (e.g., <Ctrl>-<S>).

Outside of Eclipse

Open up the AndroidManifest.xml file in an editor and add an <activity> element, as a child of the <application> element, with an android:name="SimpleContentActivity" attribute, to the file. The result should resemble:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.empublite"
>
    <activity>
        <name>SimpleContentActivity</name>
    </activity>
</manifest>
```
android:versionCode="1"
android:versionName="1.0">

<uses-sdk
    android:minSdkVersion="9"
    android:targetSdkVersion="15"/>

<supports-screens
    android:largescreens="true"
    android:normalscreens="true"
    android:smallscreens="false"
    android:xlargescreens="true"/>

<application
    android:allowbackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Sherlock.Light.DarkActionBar"
    android:uioptions="splitActionBarWhenNarrow">
    <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>
    <activity android:name="SimpleContentActivity">
    </activity>
    </application>
</manifest>

Step #3: 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:

```java
@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);
```
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.
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, including supporting fragments on pre-Android 3.0 devices.

**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.

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?

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 X1o 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. The ever-popular Angry Birds game, for example, gives you bigger birds.

However, another design approach is to consider a tablet screen to really be a collection of phone screens, side by side.

![Fig 161: 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.

**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.

**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 Project

We have two choices with fragments: use the native ones in API Level 11, or use a backport supplied by the Android Support package. So this sample can work on older versions of Android, we will use the Android Support package, adding it to the project.

We also add in ActionBarSherlock. That is not strictly required to use fragments, whether those are native API Level 11 fragments or are ones from the Android Support package. However, you may want to have an action bar in addition to fragments, in which case you would want to use ActionBarSherlock if you are using the backported fragments implementation. Also, using fragments with ActionBarSherlock requires some minor changes to your code, which this project will illustrate.

The Fragment Layout

Our fragment is going to manage our UI, so we have a res/layout/mainfrag.xml layout file containing our Button:

```xml
<?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. This class extends SherlockFragment — the Fragment implementation from ActionBarSherlock, which itself inherits from android.support.v4.app.Fragment
from the Android Support package. If you wish to use the native API Level 11 fragments, you would inherit from android.app.Fragment instead.

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:

```java
@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 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:

```java
@Override
public void onClick(View v) {
```

321
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
<?xml version="1.0" encoding="utf-8"?><fragment>
    <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"/>
</fragment>
```

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`.

Eclipse users can drag a fragment out of the “Layouts” section of the graphical editor tool palette, if desired, rather than setting up the `<fragment>` element directly in the XML.
**The Activity Class**

StaticFragmentsDemoActivity — our new launcher activity — looks identical to the previous version, with the exception of the class name:

```java
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);
    }
}
```

However, there is one change hidden in the new LifecycleLoggingActivity. We no longer inherit from Activity, but instead inherit from SherlockFragmentActivity:

```java
package com.commonsware.android.sfrag;

import android.os.Bundle;
import android.util.Log;
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class LifecycleLoggingActivity extends SherlockFragmentActivity {

    There are three primary possible base classes for your fragment-powered activities:

    1. If you are using native API Level 11 fragments and action bar, you can inherit from the ordinary Activity class as you normally would.
    2. If you are using the Android Support package for your fragments but are not using ActionBarSherlock (e.g., you are skipping an action bar on pre-API Level 11 devices), you would inherit from android.support.v4.app.FragmentActivity. This is the fragment-capable activity base class supplied by the Android Support package.
3. If you are using ActionBarSherlock, inherit from SherlockFragmentActivity.

The Result

Visually, there is no difference between this version and the previous one, except that we now have an action bar:

![A Static Fragment on Android 2.3.3](image)

*Figure 162: A Static Fragment on Android 2.3.3*

The Fragment Lifecycle Methods

Fragments have lifecycle methods, just like activities do. In fact, they support most of the same lifecycle methods as activities:

- `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.

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.

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 nonsense words as seen in previous examples.

However, since we are using ActionBarSherlock in this project, we need to use SherlockListFragment, to ensure that we will work well with the replacement action bar.

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:

```java
package com.commonsware.android.dfrag;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.widget.ArrayAdapter;
import com.actionbarsherlock.app.SherlockListFragment;

public class OtherFragment extends SherlockListFragment {


    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        setListAdapter(new ArrayAdapter<String>(getActivity(), android.R.layout.simple_list_item_1, items));
    }
}
```

We call setListAdapter() in onActivityCreated(). In principle, we could call it any time after onCreateView() is processed, such as in onResume().

---

326
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:

```java
package com.commonsware.android.dfrag;

import android.os.Bundle;

public class OtherActivity extends LifecycleLoggingActivity {
  @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 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, as we are here, 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.

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.

The Result

Our OtherActivity looks identical to the Selection/List sample from an earlier chapter, except that it sports the action bar courtesy of our ActionBarSherlock implementation:

*Figure 163: A Dynamic Fragment on Android 4.0.3*
Fragments and the Action Bar

Fragments can add items to the action bar by calling setHasOptionsMenu(true) from onActivityCreated() (or any earlier lifecycle method). This indicates to the activity that it needs to call onCreateOptionsMenu() and onOptionsItemSelected() on the fragment.

The Fragments/ActionBar sample application demonstrates this. This is the same as the ActionBar/ActionBarDemo sample from the chapter on the action bar, just with the activity converted into a dynamic fragment.

In onActivityCreated() of ActionBar fragment, we call setHasOptionsMenu(true):

```
@Override
public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    setRetainInstance(true);
    setHasOptionsMenu(true);
    if (adapter == null) {
        initAdapter();
    }
}
```

(we will discuss that setRetainInstance(true) call in a later chapter)

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);
    configureActionItem(menu);
    super.onCreateOptionsMenu(menu, inflater);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == 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:

```java
package com.commonsware.android.abf;

import android.os.Bundle;
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class ActionBarFragmentActivity extends SherlockFragmentActivity {
    @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 ActionBarFragment()).commit();
        }
    }

    // Other methods...
}
```

### 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 a new edition of the Android Support package also released in November 2012 — 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 (perhaps 2015), 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:

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.

We will see much more on this subject when we get into large-screen strategies in a later chapter.
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
- 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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book’s GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.

Step #1: Copy In WebViewFragment

Android has, as of Android 3.0, a WebViewFragment class. Just as ListFragment wraps a ListView in a Fragment, WebViewFragment wraps a WebView in a Fragment.
TUTORIAL #9 - STARTING OUR FRAGMENTS

However, for unclear reasons, WebViewFragment was not put in the Android Support package. Nor does ActionBarSherlock contain a SherlockWebViewFragment.

Fortunately, Android is open source.

So, we will incorporate a slightly-modified version of the open source WebViewFragment into our application, to use as the basis for our fragments showing book content.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

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 WebViewFragment in the “Name” field. Then, click the “Browse…” button next to the “Superclass” field and find SherlockFragment to set as the superclass. Click “Finish” on the new-class dialog to create the mostly-empty WebViewFragment.

Then, with the newly-created WebViewFragment open in the editor, replace its entire contents with the following:

```java
/*
 * Copyright (C) 2010 The Android Open Source Project
 * Portions Copyright (c) 2012 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.
*/

package android.webkit;
package com.commonsware.empublite;
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;
import com.actionbarsherlock.app.SherlockFragment;

/**
 * A fragment that displays a WebView.
 * <p>
 * The WebView is automatically paused or resumed when the
 * Fragment is paused or resumed.
 * *
 * @TargetApi(11)
 * @Override
 */
public class WebViewFragment extends SherlockFragment {
    private WebView mWebView;
    private boolean mIsWebViewAvailable;

    public WebViewFragment() {
    }

    /**
     * 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(getActivity());
        mIsWebViewAvailable = true;
        return mWebView;
    }

    /**
     * Called when the fragment is visible to the user and
     * actively running. Resumes the WebView.
     * *
     * @TargetApi(11)
     * @Override
     */
    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.
     */
package com.commonsware.empublite;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.webkit.WebView;
import android.webkit.WebViewClient;

import androidx.fragment.app.Fragment;

public class WebViewFragment extends Fragment {

  private WebView mWebView;
  private boolean mIsWebViewAvailable;

  @TargetApi(11)
  @Override
  public void onResume() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
      mWebView.onResume();
    }

    super.onResume();
  }

  @Override
  public void onDestroyView() {
    mIsWebViewAvailable=false;
    super.onDestroyView();
  }

  @Override
  public void onDestroy() {
    if (mWebView != null) {
      mWebView.destroy();
      mWebView=null;
    }
    super.onDestroy();
  }

  @Override
  public WebView getWebView() {
    return mIsWebViewAvailable ? mWebView : null;
  }
}

Outside of Eclipse

Create a src/com/commonsware/empublite/WebViewFragment.java source file, with the content shown in the code listing above.
Step #2: Examining WebViewFragment

The implementation of WebViewFragment we just created is almost identical to the one you will find in the Android open source project. Here are the highlights:

- **onCreateView()**, when first run, will create a new WebView object via its constructor, holding onto it as `mWebView`. `onCreateView()` also has an optimization to speed things up in situations such as when the screen is rotated, but the details of this are beyond the scope of this chapter.
- **onPause()** and **onResume()** invoke their corresponding methods on the WebView object. However, **onPause()** and **onResume()** were only added to the Android SDK with API Level 11. Since we want to use WebViewFragment on older devices, we use some tricks to make sure we only call **onPause()** and **onResume()** on the WebView when we are running on API Level 11 or higher. We will discuss the particular techniques shown here in an upcoming chapter on backwards compatibility.
- **onDestroyView()** sets a flag to indicate that we should no longer be using the WebView — this flag is used by the `getWebView()` method that provides the WebView to subclasses of WebViewFragment.
- **onDestroy()** calls `destroy()` on the WebView, to proactively clean up some memory that it holds.

Also, please forgive the erroneous JavaDoc comments for the **onPause()** and **onResume()** methods, which are flipped. That is the way the code appears in the Android Open Source Project, and those flaws were left intact in the backport of this class.

Step #3: Creating AbstractContentFragment

WebViewFragment is nice, but it is mostly just a manager of various lifecycle behaviors. We need to further customize the way we use that WebView widget, so we will add those refinements in another class, AbstractContentFragment.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).
TUTORIAL #9 - STARTING OUR FRAGMENTS

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 AbstractContentFragment in the “Name” field. Then, click the “Browse...” button next to the “Superclass” field and find WebViewFragment to set as the superclass — but make sure you choose the one in the com.commonsware.empublite package, not the one in android.webkit. Then, click “Finish” on the new-class dialog to create the AbstractContentFragment class.

Then, with the newly-created AbstractContentFragment open in the editor, replace its entire contents with the following:

```java
package com.commonsware.empublite;

import android.annotation.SuppressLint;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

abstract public class AbstractContentFragment extends WebViewFragment {
    abstract String getPage();

    @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().setSettings().setJavaScriptEnabled(true);
        getWebView().setSettings().setSupportZoom(true);
        getWebView().setSettings().setBuiltInZoomControls(true);
        getWebView().loadUrl(getPage());

        return(result);
    }
}
```
Outside of Eclipse

Create a `src/com/commonsware/empublite/AbstractContentFragment.java` source file, with the content shown in the code listing above.

**Step #4: Examining AbstractContentFragment**

AbstractContentFragment has but two 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 an abstract `getPage()` method. Finally, we return what the superclass returned from `onCreateView()` — effectively, we are simply splicing in our own configuration logic.

**In Our Next Episode…**

... we will set up horizontal swiping of book chapters in our tutorial project.
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 View Pager, which allows the user to swipe horizontally to move between different portions of your content. However, View Pager is not distributed as part of the firmware, but rather via the Android Support package, alongside the backport of the fragments framework. Hence, even though View Pager 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 View Pager and how to set one up.

**Swiping Design Patterns**

In 2012, Google released the Android Design Web site 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
SWIPING WITH VIEWPAGER

- Chapters in an ebook
- Tabs in a collection of tabs

The primary way to implement this pattern in Android is the `ViewPager`.

**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. Android has some built-in support for using fragments inside of `ViewPager` that make it fairly easy to use.

To see this in action, this section will examine the `ViewPager/ Fragments` sample project.

**The Prerequisites**

The project has a dependency on the Android Support package, in order to be able to use `ViewPager`. And, as do most of this book’s samples from this point forward, it also depends upon ActionBarSherlock, so we can have an action bar while still supporting Android 2.1 and beyond.

**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
<?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>
```

**The Activity**

As you see, the `ViewPagerFragmentDemoActivity` itself is blissfully small:

```java
package com.commonsware.android.pager;

import android.os.Bundle;
import android.support.v4.view.ViewPager;
```
All we do is load the layout, retrieve the ViewPager via findViewById(), and provide a SampleAdapter to the ViewPager via setAdapter().

The PagerAdapter

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.

For the moment, we will focus on FragmentPagerAdapter.

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)
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.

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:

```xml
    android:id="@+id/editor"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:inputType="textMultiLine"
    android:gravity="left|top"
/>```

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):

```java
static EditorFragment newInstance(int position) {
    EditorFragment frag = new EditorFragment();
    Bundle args = new Bundle();

    args.putInt(KEY_POSITION, position);
    frag.setArguments(args);
}
```
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`:

```java
@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:

![Figure 164: A ViewPager on Android 4.0.3](image-url)
However, you can horizontally swipe to get to the next fragment:

![Figure 165: A ViewPager in Use on Android 4.0.3](image)

Swiping works in both directions, so long as there is another fragment 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.
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 android:id="@+id/pager"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <android.support.v4.view.PagerTitleStrip
        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.
Our SampleAdapter needs another method: `getPageTitle()`, which will return the title to display in the PagerTabStrip for a given position:

```java
package com.commonsware.android.pager2;

import android.content.Context;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.Fragment;
import android.support.v4.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:

```java
package com.commonsware.android.pager2;

import android.content.Context;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;
import com.actionbarsherlock.app.SherlockFragment;

public class EditorFragment extends SherlockFragment {
    private static final String KEY_POSITION="position";

    static EditorFragment newInstance(int position) {
        return(new EditorFragment().setArguments(new Bundle().putString(KEY_POSITION, 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);
}
Note that PagerTabStrip was added after the original version of the Android Support package. If you are encountering problems finding PagerTabStrip, you may be using an older copy of the Android Support package (e.g., one that may have shipped with ActionBarSherlock).

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](https://commonsware.com), brought to you by the author of ActionBarSherlock. 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](https://commonsware.com).

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 natively, and the latest Android Support package backport, does support nested fragments. Now 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 our PagerAdapter, as is illustrated in the [ViewPager/Nested](https://commonsware.com) 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:

```java
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();
        }
    }
}
```

350
This loads a PagerFragment, which contains most of the logic from our original activity:

```java
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(). 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:
SWIPE WITH VIEWPAGER

- 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;
}
```

- 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.
**SWIPING WITH **

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 167: A ViewPager, PagerTabStrip, and Action Bar Item on Android 4.1](image)

...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)`:

---

353
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.
SWIPING WITH VIEWPAGER

We will see an example of this later in the book, when we put some maps into a ViewPager.
Tutorial #10 - Rigging Up a ViewPagerAdapter

A ViewPagerAdapter is a fairly slick way to present a digital book. You can have individual chapters be accessed by horizontal swiping, with the prose within a chapter 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 ViewPagerAdapter 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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book’s GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.

Step #1: Add a ViewPagerAdapter to the Layout

Right now, the layout for EmPubLiteActivity just has a Progressbar. We need to augment that to have our ViewPagerAdapter as well, set up such that we can show either the Progressbar (while we load the book) or the ViewPagerAdapter as needed.

Unfortunately, this is the sort of change that the Eclipse drag-and-drop GUI building is not particularly well-suited for. Hence, even Eclipse users are going to have to dive into the layout XML this time.
TUTORIAL #10 - RIGGING UP A VIEWPAGER
Open up res/layout/main.xml (and, if you are using Eclipse, switch to the
“main.xml” sub-tab of the editor, to see the raw XML). As a child of the
<RelativeLayout>, after the <ProgressBar>, add a
<android.support.v4.view.ViewPager> element as follows:
<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
null;

358

Subscribe to updates at https://commonsware.com

Special Creative Commons BY-NC-SA 4.0 License Edition


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:

```java
pager=(ViewPager)findViewById(R.id.pager);
```

If you are using Eclipse, you will 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 PagerView 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.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**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 to set as the superclass. 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:

```java
public ContentsAdapter(SherlockFragmentActivity ctxt) {
    super(ctxt.getSupportFragmentManager());
}
```

This simply chains to the superclass, supplying the requisite FragmentManager instance, culled from our parent activity.
You will need to import `com.actionbarsherlock.app.SherlockFragmentActivity` for this to compile.

### Outside of Eclipse

Create a `src/com/commonsware/empublite/ContentsAdapter.java` source file, with the following content:

```java
package com.commonsware.empublite;

import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentStatePagerAdapter;
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class ContentsAdapter extends FragmentStatePagerAdapter {
    public ContentsAdapter(SherlockFragmentActivity ctxt) {
        super(ctxt.getSupportFragmentManager());
    }

    @Override
    public Fragment getItem(int position) {
        return null;
    }

    @Override
    public int getCount() {
        return 0;
    }
}
```

### 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`:

```java
adapter=new ContentsAdapter(this);
pager.setAdapter(adapter);
findViewById(R.id.progressBar1).setVisibility(View.GONE);
findViewById(R.id.id.pager).setVisibility(View.VISIBLE);
```

This will require a new data member:

```java
private ContentsAdapter adapter=null;
```

It will also require an import for `android.view.View`.

---

360

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 View Pager, and toggling the visibility of the ProgressBar (making it GONE) and the View Pager (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 View Pager:

![Figure 169: EmPubLite, With Empty View Pager](image)

The View Pager is empty simply because our ContentsAdapter returned 0 from getCount(), indicating that there are no pages to be displayed.

**In Our Next Episode…**

... we will finish our “help” and “about” screens in our tutorial project.
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)
• Different number and capabilities of cameras
• 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? Is the screen square and, therefore, does not really have an orientation?
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. **Touchscreen**: does the device have a touchscreen? If so, is the touchscreen set up to be used with a stylus or a finger?
5. **Keyboard**: what keyboard does the user have (QWERTY, numeric, neither), either now or as an option?
   6. **Other input**: does the device have some other form of input, like a directional pad or click-wheel?

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 would 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.

An even 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 `@ orientation if you prefer.

Screen size is a bit more complicated, simply because the available approaches have changed over the years.

**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

So, a `res/layout-small/` directory would hold resources related to small-screen devices. The convention was to put `-normal` resources in default directories (e.g., `res/layout/`) and use the resource set qualifiers for the other buckets as needed. For maximum backwards compatibility, though, Android will “cheat” in one case: if you have a `-large` resource set, but no `-xlarge` resource set, an `-xlarge` device will use `-large` instead of the default set.

**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”. Since 1 dp is 1/160th of an inch, those shortest widths equate to 560–600 dp. Hence, you might set up a -sw600dp 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 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:

```xml
<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/
(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 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
**Resource Sets and Configurations**

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.

**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.

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

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.

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/.

**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.

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.

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).

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

The best approach nowadays 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.

The activity itself just loads the dynamic fragment, following the recipe seen previously in this book:

```java
package com.commonsware.android.rotation.frag;
import android.os.Bundle;
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class RotationFragmentDemo extends SherlockFragmentActivity {
    @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 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
<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>
```

The portrait edition, in res/layout/, is identical save for the orientation of the LinearLayout:

```xml
<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"
```

Here is the complete implementation of RotationFragment:

```java
package com.commonsware.android.rotation.frag;

import android.app.Activity;
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;
import com.actionbarsherlock.app.SherlockFragment;

public class RotationFragment extends SherlockFragment 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);
        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:
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`.
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.

**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 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 stick 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.

**Add to the Bundle**

However, you may not be using fragments, in which case setRetainInstance(true) will not be available to you. In that case, you will have to turn to some alternative approaches.

The best of those is 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:

- 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:

```java
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    if (contact != null) {
        outState.putString("contact", contact.toString());
    }
}

@Override
protected void onRestoreInstanceState(Bundle state) {
    super.onRestoreInstanceState(state);
    String contactUri = state.getString("contact");
    if (contactUri != null) {
        contact = Uri.parse(contactUri);
        viewButton.setEnabled(true);
    }
}
```

The big benefit of this approach is that `onSaveInstanceState()` is used for another scenario, beyond configuration changes.

Suppose, while the user is using one of your activities, a text message comes in. The user taps on the notification and goes into the text messaging client, while your activity is paused and stopped. While texting, the other party sends over a URL in one of the messages. The user taps on that URL to open up a Web browser. And, right at that moment, a phone call comes in.

Android may not have enough free RAM to handle launching the browser and the phone applications, because too many things are happening at once. Hence, Android may terminate your process, to free up RAM. Yet, it is entirely possible that the user could return to your activity via the BACK button.

If the user does return to your activity via BACK, Android will fork a fresh process for your application, will create a new instance of your activity, and will supply to that activity the `Bundle` from `onSaveInstanceState()` of the old activity. This way, you can help retain context from what the user had been doing, despite your entire process having been gone for a while.
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()`
- `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:

```java
@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    if (contact != null) {
        outState.putString("contact", contact.toString());
    }
}
```

Our `onCreateView()` method examines the passed-in Bundle, and if it is not null tries to obtain our contact from it:

```java
@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) {
        String contactUri=state.getString("contact");
        if (contactUri != null) {
            contact=Uri.parse(contactUri);
        }
    }
    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).

### Retain Other Objects

The problem with onSaveInstanceState() is that you are limited to a Bundle. That’s because this callback is also used in cases where your whole process might be terminated (e.g., low memory), so the data to be saved has to be something that can be serialized and has no dependencies upon your running process.

For some activities, that limitation is not a problem. For others, though, it is more annoying. Take an online chat, for example. You have no means of storing a socket in a Bundle, so by default, you will have to drop your connection to the chat server and re-establish it. That not only may be a performance hit, but it might also affect the chat itself, such as you appearing in the chat logs as disconnecting and reconnecting.

One way to get past this is to use onRetainNonConfigurationInstance() instead of onSaveInstanceState() for “light” changes like a rotation. Your activity’s onRetainNonConfigurationInstance() callback can return an Object, which you can retrieve later via getLastNonConfigurationInstance(). The Object can be just about anything you want — typically, it will be some kind of “context” object holding activity state, such as running threads, open sockets, and the like. Your activity’s onCreate() can call getLastNonConfigurationInstance() — if you get a non-null response, you now have your sockets and threads and whatnot.

The biggest limitation is that you do not want to put in the saved context anything that might reference a resource that will get swapped out, such as a Drawable loaded from a resource.
The second-biggest limitation is that you do not want to put in the saved context anything that has a reference back to your original activity instance. Otherwise, the new activity will hold an indirect reference back to the old activity, and the old activity will not be able to be garbage-collected.

The general strategy, therefore, is to use onSaveInstanceState() for everything that it can handle, since it covers other scenarios beyond configuration changes. Use onRetainNonConfigurationInstance() for everything else.

To see this approach, take a look at the ConfigChange/Retain sample project.

This is the same as the previous sample, except that RotationRetainDemo implements onRetainNonConfigurationInstance(), returning the Uri that represents our selected contact:

```java
@Override
public Object onRetainNonConfigurationInstance() {
    return (contact);
}
```

In onCreate(), we call getLastNonConfigurationInstance(). This will either be null or our Uri from a preceding instance. In either case, we store the value in contact and use it:

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    viewButton=(Button)findViewById(R.id.view);
    contact=(Uri)getLastNonConfigurationInstance();
    viewButton.setEnabled(contact != null);
}
```

**DIY**

In a few cases, even onRetainNonConfigurationInstance() 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:
**Resource Sets and Configurations**

- 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:

```xml
<activity
    android:name="RotationDIYDemo"
    android:configChanges="keyboardHidden|orientation|screenSize"
    android:label="@string/app_name">
    <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 `smallestScreenSize`, if you have your `android:targetSdkVersion` set to 13 or higher. Note that this will require your build target 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:

```java
@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);
    }
}
```
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.

It is these two problems that are why Google does not recommend the use of this technique unless absolutely necessary.

**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="landscape", or possibly android:screenOrientation="portrait", to your manifest.

Ideally, you choose landscape, as some devices (e.g., Google 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).
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.
DEALING WITH THREADS

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.

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).

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. The pre-eminent 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, Android developers who have struggled with background thread management do not strictly want background threads — they want work to be done off the UI thread, so users are not stuck waiting and activities do not get the dreaded “application not responding” (ANR) error. And while Android cannot magically cause work to not 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, commonly as a private inner class of something that uses the task (e.g., an activity)
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
DEALING WITH THREADS

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.

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.
DEALING WITH THREADS

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.

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
DEALING WITH THREADS

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 SherlockListFragment, named AsyncDemoFragment:

```java
package com.commonsware.android.async;

import java.util.ArrayList;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.SystemClock;
import android.widget.ArrayAdapter;
import android.widget.Toast;
import com.actionbarsherlock.app.SherlockListFragment;

public class AsyncDemoFragment extends SherlockListFragment {
    private ArrayList<String> model=null;
    private ArrayAdapter<String> adapter=null;
    private AddStringTask task=null;

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        setRetainInstance(true);

        if (model == null) {
            model=new ArrayList<String>();
            task=new AddStringTask();
            task.execute();
        }

        adapter=
```

394
new ArrayAdapter<String>(getActivity(),
android.R.layout.simple_list_item_1,
model);

g.getListView().setScrollbarFadingEnabled(false);
setListAdapter(adapter);
}

@Override
public void onDestroy() {
    if (task != null) {
        task.cancel(false);
    }
    super.onDestroy();
}

class AddStringTask extends AsyncTask<Void, String, Void> {
    @Override
    protected Void doInBackground(Void... unused) {
        for (String item : items) {
            if (!isCancelled()) {
                publishProgress(item);
                SystemClock.sleep(400);
            }
        }
        return(null);
    }

    @Override
    protected void onProgressUpdate(String... item) {
        adapter.add(item[0]);
    }

    @Override
    protected void onPostExecute(Void unused) {
        if (!isCancelled()) {
            Toast.makeText(getActivity(), R.string.done, Toast.LENGTH_SHORT)
                .show();
        }
        task=null;
    }
}

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.
DEALING WITH THREADS

In `onActivityCreated()`, 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. We then 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`.

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.
DEALING WITH THREADS

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, if our task has not been cancelled (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.

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 SherlockFragmentActivity with the standard recipe for kicking off an instance of a dynamic fragment:

```java
package com.commonsware.android.async;

import android.os.Bundle;
import com.actionbarsherlock.app.SherlockFragmentActivity;
```
DEALING WITH THREADS

```java
public class AsyncDemo extends SherlockFragmentActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getSupportFragmentManager().findFragmentById(R.id.content) == null) {
            getSupportFragmentManager().beginTransaction().add(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 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 nonsense 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 nonsense 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()`.

Why?

Most callback methods in Android are driven by messages on the message queue being processed by the main application thread. Normally, this queue is being processed whenever the main application thread is not otherwise busy, such as running our code.

However, when a configuration change occurs, like a screen rotation, that no longer holds true.

Android guarantees that, while on the main application thread, `getActivity()` will return a valid `Activity`. Moreover, once the configuration change starts, no messages on the message queue will be processed until after `onCreate()` of the hosting activity (and `onActivityCreated()` of the fragment) have completed their work.

**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.

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 4.0, 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. We will examine this more in a later chapter on dealing with backwards-compatibility issues.

**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`:

```java
package com.commonsware.android.post;

import android.app.Activity;
```

```
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 in the `java.util.concurrent` package that will help you communicate safely with your background thread.
2. The possibility that the activity will be killed off while background work is going on. For example, after starting your activity, the user might have a call come in, followed by a text message, followed by a need to look up a contact… all of which might be sufficient to kick your activity out of memory.
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.
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.
REQUESTING PERMISSIONS

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:

```xml
<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.
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 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.

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.

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 your 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 pre-installed on the firmware.

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:

```xml
<permission>
  android:name="android.permission.BRICK"
  android:label="@string/permlab_brick"
  android:description="@string/permdesc_brick"
  android:protectionLevel="signature" />
</permission>

<permission>
  android:name="android.permission.REBOOT"
  android:label="@string/permlab_reboot"
  android:description="@string/permdesc_reboot"
  android:protectionLevel="signatureOrSystem" />
</permission>
```

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 signatureOrSystem, meaning that the app could just be installed as part of the firmware to hold this permission.

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.
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.
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...
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, 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.

**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 `OutputStream` 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).

On Android 1.x and 2.x, the big advantage of external storage is size. Some Android devices have very little internal storage (tens or hundreds of MB) that *all apps* must share. External storage, on the other hand, typically is on the order of GB of
available free space. On Android 3.0+, internal and external storage are usually implemented via the same filesystem partition, and therefore both tend to be large.

However, on Android 1.x and 2.x, external storage is not always available – 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.

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:

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/location"
    android:actionLayout="@layout/action_location"
    android:showAsAction="always"/>
  <item
    android:id="@id/save"
    android:icon="@android:drawable/ic_menu_save"
    android:showAsAction="always|withText"
    android:title="@string/save"/>
  <item
    android:id="@id/saveBackground"
    android:icon="@android:drawable/ic_menu_save"
    android:showAsAction="never"
    android:title="@string/saveBackground"/>
</menu>
```

We get at that `CheckBox` in `onCreateOptionsMenu()` of `EditorFragment`, storing it in a data member of the fragment:

```java
@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()`:

```java
private File getTarget() {
  File root = null;
```
if (external != null && external.isChecked()) {
    root = getActivity().getExternalFilesDir(null);
} else {
    root = getActivity().getFilesDir();
}

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.

**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:

```java
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));
}

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:

- 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
<?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="9"
    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"/>
```
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 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`:

```java
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, 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, StackOverflow 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 (and perhaps future versions), 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. Many devices still use YAFFS today.
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 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:

```java
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 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 building a debug version of the app and are on a version of Android that supports StrictMode:

```java
package com.commonsware.android.frw;
import android.os.Build;
import android.os.Bundle;
import android.os.StrictMode;
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class FilesDemoActivity extends SherlockFragmentActivity {
    @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()).

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:
04-19 11:13:41.522: D/StrictMode(1443):   at android.os.StrictMode$AndroidBlockGuardPolicy.onReadFromDisk(StrictMode.java:1089)
04-19 11:13:41.522: D/StrictMode(1443):   at libcore.io.BlockGuardOs.open(BlockGuardOs.java:106)
04-19 11:13:41.522: D/StrictMode(1443):   at com.commonsware.android.frw.EditorFragment.onOptionsItemSelected(EditorFragment.java:73)
04-19 11:13:41.522: D/StrictMode(1443):   at com.android.internal.policy.impl.PhoneWindow.onMenuItemSelected(PhoneWindow.java:950)
04-19 11:13:41.522: D/StrictMode(1443):    at com.android.internal.policy.impl.PhoneWindow.onMenuItemSelected(PhoneWindow.java:950)
dalvik.system.NativeStart.main(Native Method)
-durati0n=1 ms: android.os.StrictMode$StrictModeDiskWriteViolation:
   policy=31 violation=1
android.os.StrictMode$AndroidBlockGuardPolicy.onWriteToDisk(StrictMode.j
ava:1063)
libcore.io.BlockGuardOs.fsync(BlockGuardOs.java:96)
java.io.FileDescriptor.sync(FileDescriptor.java:71)
com.commonsware.android.frw.EditorFragment.save(EditorFragment.java:111)
com.commonsware.android.frw.EditorFragment.onOptionsItemSelected(EditorF
ragment.java:73)
com.actionbarsherlock.app.SherlockFragment.onOptionsItemSelected(Sherloc
kFragment.java:67)
android.support.v4.app.FragmentManagerImpl.dispatchOptionsItemSelected(F
ragmentManager.java:1919)
com.actionbarsherlock.app.SherlockFragmentActivity.onMenuItemSelected(Sh
erlockFragmentActivity.java:288)
com.actionbarsherlock.ActionBarSherlock.callbackOptionsMenuSelected(Acti
onBarSherlock.java:586)
com.actionbarsherlock.Activity.actionBarSherlockInternal.dispatchOptionsIt
emSelected(ActivityBarSherlockNative.java:78)
com.actionbarsherlock.app.SherlockFragmentActivity.onMenuItemSelected(Sh
erlockFragmentActivity.java:191)
com.android.internal.policy.impl.PhoneWindow.onMenuItemSelected(PhoneWin
dow.java:950)
com.android.internal.view.menu.MenuBuilder.dispatchMenuItemSelected(Menu
Builder.java:735)
com.android.internal.view.menu.MenuItemImpl.invoke(MenuItemImpl.java:149
)
com.android.internal.view.menu.MenuBuilder.performMenuItemAction(MenuBuilde
r.java:874)
com.android.internal.view.menu.ActionMenuView.invokeItem(ActionMenuView.
java:490)

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.

## 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. The best third-party option today 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.
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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book’s GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.

Step #1: Adding Some Content

Your project should already have an assets/ folder. If not, create one. In Eclipse, you would do this by right-clicking over the project in the Package Explorer, choosing New > Folder from the context menu, filling in the name assets in the dialog, and clicking “Finish”.

In assets/, create a misc/ sub-folder — Eclipse users would use the same technique as above, but start by right-clicking over the assets/ folder instead of the project.

In assets/misc/, create two files, about.html and help.html. 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: Create a SimpleContentFragment**

Now, we need to arrange to load this content. `WebViewFragment` and `AbstractContentFragment` are fine and all, but neither know how to actually load anything. In `AbstractContentFragment`, this is handled by `getPage()`, which is an abstract method. So, let's create a `SimpleContentFragment` subclass of `AbstractContentFragment` that knows how to load files out of our project's assets.

If you wish to make this change using Eclipse's wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**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 `SimpleContentFragment` in the “Name” field. Then, click the “Browse...” button next to the “Superclass” field and find `AbstractContentFragment` to set as the superclass. Then, click “Finish” on the new-class dialog to create the `SimpleContentFragment` class.

Then, replace its contents with the following:

```java
package com.commonsware.empublite;

import android.os.Bundle;

public class SimpleContentFragment extends AbstractContentFragment {
    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
String getPage() {
    return(getArguments().getString(KEY_FILE));
}

Outside of Eclipse

Create a src/com/commonsware/empublite/SimpleContentFragment.java source file, with the content shown above.

Step #3: Examining SimpleContentFragment

SimpleContentFragment does indeed override our getPage() abstract method. What it returns is a value out of the “arguments” Bundle supplied to the fragment — specifically the string identified as KEY_FILE.

SimpleContentFragment sets up those arguments via 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.

This means that anyone wanting to use SimpleContentFragment should use the factory method, to provide the path to the content to load.

Step #4: Using SimpleContentFragment

Now, we need to use this fragment in an activity somewhere. We already set up a stub SimpleContentActivity for this purpose, but we left its implementation completely empty.

Now, open up SimpleContentActivity and fill in the following Java:

package com.commonsware.empublite;
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 #5: 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:

```java
@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);
```
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:

```
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);
}
```

Figure 173: EmPubLite Help Screen
Pressing BACK and choosing “About” from the action bar overflow will bring up your about content:

![About Screen](image)

*Figure 174: EmPubLite About Screen*

In Our Next Episode…

... we will display the actual content of our book in our tutorial project.
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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book’s GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.

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.
Download http://misc.commonsware.com/WarOfTheWorlds.zip and unpack its contents (a book/directory of files) into your assets/ folder of your project. Eclipse users can drag this book/directory into the Package Explorer 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: 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”:

```json
{
    "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"
        }
    ]
}
```

In the case of this book from Project Gutenberg, the assets/book/ directory contains five 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 org.json parsing classes.

If you wish to make this change using Eclipse's wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**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 following class definition:

```java
package com.commonsware.empublite;

import org.json.JSONArray;
import org.json.JSONObject;

public class BookContents {
  JSONObject raw=null;
  JSONArray chapters;

  BookContents(JSONObject raw) {
    this.raw=raw;
    chapters=raw.optJSONArray("chapters");
  }

  int getChapterCount() {
    return(chapters.length());
  }

  String getChapterFile(int position) {
    JSONObject chapter=chapters.optJSONObject(position);

    return(chapter.optString("file"));
  }
}
```
String getTitle() {
    return raw.optString("title");
}

Outside of Eclipse

Create a src/com/commonsware/empublite/BookContents.java source file, with the content shown above.

Step #3: Examining Our Model

Our BookContents constructor takes a JSONObject parameter. This will be supplied by some other code that we have not yet written, and it will contain the entire contents.json structure. JSONObject behaves a bit like the XML DOM, in that it holds the entire parsed content in memory.

BookContents is partially a wrapper around the JSONObject, offering getters for specific bits of information, notably:

- getChapterCount() to identify the number of chapters (i.e., the size of the JSONArray created from our 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 #4: Creating a ModelFragment

Something has to load that BookContents, ideally in the background, since reading an asset and parsing the JSON will take time.

Something has to hold onto that BookContents, so it can be used from EmPubLiteActivity and the various chapter fragments in the ViewPager.

In our case, we will use the “model fragment” approach outlined in a previous chapter, with a new class, cunningly named ModelFragment.

If you wish to make this change using Eclipse's wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).
**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 `SherlockFragment` to set as the superclass. Then, click “Finish” on the new-class dialog to create the `ModelFragment` class.

Finally, paste in the following definition for `ModelFragment`:

```java
tutorial #12 - displaying the book

package com.commonsware.empublite;

import android.annotation.TargetApi;
import android.content.Context;
import android.os.AsyncTask;
import android.os.Build;
import android.os.Bundle;
import android.util.Log;
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import com.actionbarsherlock.app.SherlockFragment;
import org.json.JSONObject;

public class ModelFragment extends SherlockFragment {
    private BookContents contents = null;
    private ContentsLoadTask contentsTask = null;

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        setRetainInstance(true);
        deliverModel();
    }

    synchronized private void deliverModel() {
        if (contents != null) {
            ((EmPubLiteActivity)getActivity()).setUpPager(contents);
        } else {
            if (contents == null && contentsTask == null) {
                contentsTask = new ContentsLoadTask();
                executeAsyncTask(contentsTask, getActivity().getApplicationContext());
            }
        }
    }

    @TargetApi(11)
    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);
    }
}

private class ContentsLoadTask extends AsyncTask<Context, Void, Void> {
    private BookContents localContents = null;
    private Exception e = null;

    @Override
    protected Void doInBackground(Context... ctxt) {
        try {
            StringBuilder buf = new StringBuilder();
            InputStream json = ctxt[0].getAssets().open("book/contents.json");
            BufferedReader in =
                new BufferedReader(new InputStreamReader(json));
            String str;
            while ((str = in.readLine()) != null) {
                buf.append(str);
            }
            in.close();
            localContents = new BookContents(new JSONObject(buf.toString()));
        } catch (Exception e) {
            this.e = e;
        }
        return (null);
    }

    @Override
    public void onPostExecute(Void arg0) {
        if (e == null) {
            ModelFragment.this.contents = localContents;
            ModelFragment.this.contentsTask = null;
            deliverModel();
        } else {
            Log.e(getClass().getSimpleName(), "Exception loading contents",
                 e);
        }
    }
}
Note that Eclipse is going to complain that a non-existent `setUpPager()` is being called, but we will fix that later in this chapter.

**Outside of Eclipse**

Create a `src/com/commonsware/empublite/ModelFragment.java` source file, with the content shown above.

**Step #5: Examining the ModelFragment**

The point behind `ModelFragment` is to load our data (asynchronously) and hold onto it, using the retained-fragment pattern.

The catch is that even though we are retaining the fragment and holding onto the model data, the activity housing this fragment will still be destroyed and recreated on a configuration change, like a screen rotation. So, the first time our fragment is used, we need to load the content; the second and subsequent times the fragment is used, we need to simply hand over the already-loaded content. Combine that with the (slight) possibility that the user might rotate the screen before we completed loading the content the first time, and things can get a wee bit complicated.

`ModelFragment` overrides `onActivityCreated()`, to get control once `EmPubLiteActivity` has created the `ViewPager` and so on. Here, we call `setRetainInstance(true)`, so the work we do to load the `BookContents` does not evaporate, and we call a `deliverModel()` method.

The `deliverModel()` method is responsible for determining if we have our model data, handing that over to the activity (via `setUpPager()`) if we do, or starting a `ContentsLoadTask` if we do not.

Starting a `ContentsLoadTask` is delegated to a static `executeAsyncTask()` method that is designed to work around the limitation established in API Level 14, where `AsyncTask` becomes serialized, with only one task executing at a time. While we only have one task at the moment, that will change soon enough. And, while we have our `android:targetSdkVersion` set to 11, and therefore the serialized `AsyncTask` behavior should not take effect (that requires a value of 14 or higher), it is good form to start addressing this sooner rather than later. The details of what `executeAsyncTask()` is doing and how it is doing it will be covered in a later chapter.
You may notice that we are calling executeAsyncTask() with a parameter of getApplicationContext(). getApplicationContext(). getApplicationContext() returns a singleton Context object (actually an instance of an Application). This is useful in cases where we need a Context that will be around all the time. It is unsafe for us to reference an Activity in a background thread, as the Activity could conceivably be destroyed while the thread is in operation. The Application will not be destroyed so long as our process is running, so it is safer to use from a background thread.

The ContentsLoadTask itself is an AsyncTask, much akin to others we have seen so far in this book. In doInBackground(), we read in assets/book/contents.json by means of an AssetManager (obtained from the Context via getAssets()) and its open() method. This returns an InputStream, which we stream into a StringBuilder. We then parse that as JSON using JSONObject, passing the result into a BookContents instance.

In onPostExecute(), we take advantage of the fact that this is called on the main application thread, meaning we are not executing anything else on the main application thread at the time. So, it is safe for us to update our contents and contentsTask data members, plus trigger a call to deliverModel(), which will pass the BookContents along to the EmPubLiteActivity. If something went wrong during the JSON load, and we had an exception, doInBackground() saves that in a data member of the ContentsLoadTask. onPostExecute() could arrange to display the error message to the user — for simplicity, we are only logging it to LogCat at the moment.

**Step #6: Supplying the Content**

Now, we need to add that missing setupPager() method on EmPubLiteActivity. Define the method, taking a BookContents as a parameter, and returning void:

```java
void setupPager(BookContents contents) {
}
```

Move these four lines from onCreate() to setupPager():

```java
adapter=new ContentsAdapter(this);
pager.setAdapter(adapter);
findViewById(R.id.progressBar1).setVisibility(View.GONE);
findViewById(R.id pager).setVisibility(View.VISIBLE);
```

`444`
Finally, pass the BookContents to the ContentsAdapter constructor as the second parameter, despite the fact that Eclipse will complain because we have not implemented that yet (we will shortly). You should wind up with a setupPager() that resembles:

```java
void setupPager(BookContents contents) {
    adapter = new ContentsAdapter(this, contents);
    pager.setAdapter(adapter);

    findViewById(R.id.progressBar1).setVisibility(View.GONE);
    findViewById(R.id.pager).setVisibility(View.VISIBLE);
}
```

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:

```java
private static final String MODEL = "model";
```

Then, modify onCreate() to see if we already have the fragment before creating one:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getSupportFragmentManager().findFragmentByTag(MODEL) == null) {
        getSupportFragmentManager().beginTransaction()
            .add(new ModelFragment(), MODEL)
            .commit();
    }

    setContentView(R.layout.main);

    pager = (ViewPager) findViewById(R.id.pager);
}
```

## Step #7: Adapting the Content

Finally, we need to update ContentsAdapter to actually use the BookContents and display the prose on the screen.

First, add a BookContents data member to ContentsAdapter:

```java
private BookContents contents = null;
```
Then, add the BookContents parameter to the constructor, assigning it to the new data member:

```java
public ContentsAdapter(SherlockFragmentActivity ctxt,
                      BookContents contents) {
    super(ctxt.getSupportFragmentManager());
    this.contents = contents;
}
```

Next, update getCount() to use the getChapterCount() of our BookContents:

```java
@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:

```java
@Override
public Fragment getItem(int position) {
    String path = contents.getChapterFile(position);

    return (SimpleContentFragment.newInstance("file:///android_asset/book/
                                               + path));
}
```

If you run the result in a device or emulator, you will see the book content appear:
Swiping left and right will take you to the other chapters in the book.

**Step #8: 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 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:

```java
    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:
getSupportActionBar().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.

**In Our Next Episode…**

... we will allow the user to [manipulate some preferences](https://commonsware.com) in our tutorial project.
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
Using Preferences

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.
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
<?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"/>
    </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:
package com.commonsware.android.preffragsbc;

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;
import com.actionbarsherlock.app.SherlockFragment;

public class PreferenceContentsFragment extends SherlockFragment {

    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.
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:

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:
On a phone-sized screen, those panes become two separate screens, the first showing the list of headers:
and the second showing the contents of a specific header:

Figure 178: Android 4.0 PreferenceActivity, on Phone
On Android 1.x and 2.x, where preference headers do not exist, we will instead show all of the preferences in one long list:
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`:

```xml
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">

    <CheckBoxPreference
        android:key="checkbox2"
        android:summary="@string/pref5summary"
        android:title="@string/pref5title"/>

```

Figure 180: Android 2.3.3 PreferenceActivity
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 three 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
3. `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
2. `android:title`, which is a few words identifying this header to the user
3. 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 FragmentActivity implementation — named EditPreferences — has a pair of static inner classes representing the preference fragments:

```java
@Override
protected boolean isValidFragment(String fragmentName) {
    if (First.class.getName().equals(fragmentName)
        || Second.class.getName().equals(fragmentName)) {
        return true;
    }
    return false;
}

@TargetApi(Build.VERSION_CODES.HONEYCOMB)
public static class First extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

Each of those PreferenceFragment implementations has an onCreate() method that calls addPreferencesFromResource() for its specific preferences.
We will illustrate the other approach, of creating a reusable PreferenceFragment, later in this chapter.

**Creating Your PreferenceActivity**

In an ideal world, the Android Support package would have an implementation of PreferenceActivity that uses preference headers and supports older versions of Android. In an ideal world, authors of Android books would have great hair. Hence, it is not an ideal world.

This causes some difficulty, insofar as API Level 11's PreferenceActivity would really like to use preference headers, and previous API levels do not support them at all.

Hence, we have to get a bit creative in our EditPreferences implementation of PreferenceActivity:

```java
package com.commonsware.android.preffragsbc;

import java.util.List;
import android.annotation.TargetApi;
import android.os.Build;
import android.os.Bundle;
import android.preference.PreferenceFragment;
import com.actionbarsherlock.app.SherlockPreferenceActivity;

public class EditPreferences extends SherlockPreferenceActivity {
    @SuppressWarnings("deprecation")
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
            addPreferencesFromResource(R.xml.preferences);
            addPreferencesFromResource(R.xml.preferences2);
        }
    }

    @TargetApi(Build.VERSION_CODES.HONEYCOMB)
    @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);
        }
    }
}
```
Our `onCreate()` entry point is called no matter what version of Android we are running on. However, for API Level 11+, there is a different callback, `onBuildHeaders()`, that we use to supply the preference headers, via a call to `loadHeadersFromResource()`.

`onBuildHeaders()` will only be called on API Level 11 and higher. Hence, there is no danger in having that method exist on older devices — it will simply be ignored.

However, on older devices, we must arrange to set up the preferences some other way. The original way to define preferences for a `PreferenceActivity` was to call `addPreferencesFromResource()`, once for each preference XML file, identifying the preferences to load. Hence, we have a pair of `addPreferencesFromResource()` calls in `onCreate()` to load our preference XML.

However, we do not want to go through that code block if we are on API Level 11+, as we will wind up with duplicated preferences: one set from the `addPreferencesFromResource()` calls and one set from the `onBuildHeaders()` logic. Hence, we wrap the `addPreferencesFromResource()` calls in a version guard block. The `android.os.Build` class has an inner class named `VERSION`, which itself has a static data member named `SDK_INT`, which returns the API level that the device is running. We can compare this to `Build.VERSION_CODES.HONEYCOMB` to see if we are...
on API Level 11 or something older, and only use addPreferencesFromResource() if we are on older devices.

We will see this version guard block technique in greater detail in a later chapter.

But, the net result is that our PreferenceActivity loads up the preferences to show to the user, using the preference header style on API Level 11 and up, and using a single list of preferences on older versions of Android.

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., 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 4.0.3.

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.

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
using preferences

android:key, you can include any attribute understood by EditText, such as
android:inputType.

The value stored in the SharedPreferences is a string.

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
relat
elated

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.

For
example,
the
sample
application
profiled
in
this
chapter
has
a
pair
of
string
array
resources
in
res/values/arrays.xml:

```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
game
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:

```xml
<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:

465
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.
USING PREFERENCES
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
467

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:

```java
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:

```java
package com.commonsware.android.pref1header;

import java.util.List;
import android.annotation.TargetApi;
import android.os.Build;
import android.os.Bundle;
import com.actionbarsherlock.app.SherlockPreferenceActivity;

public class EditPreferences extends SherlockPreferenceActivity {
  private boolean needResource=false;

  @SuppressWarnings("deprecation")
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (needResource
        || Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
```
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, or if we are on API Level 10 or below, 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:

```java
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 = getPreferenceFragment().getResources()
        .getIdentifier(getArguments().getString("resource"),
                      "xml",
                      getActivity().getPackageName());

    addPreferencesFromResource(res);
  }

  @TargetApi(Build.VERSION_CODES.HONEYCOMB)
  @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));
  }
}
```

469
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:

```xml
<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.

### 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. On the plus side, this approach is better documented and therefore perhaps more stable. However, it requires you to have your own rules for whether or not the master-detail perspective is likely to be seen.

To see how this works, take a look at the `Prefs/SingleHeader2` sample project.

To determine whether or not we wish to show the preference headers, we use a boolean resource, `R.bool.suppressHeaders`, defined both in `res/values/bools.xml`:

```xml
<resources>
  <bool name="suppressHeader">true</bool>
</resources>
```

and in `res/values-large/bools.xml`:

```xml
<resources>
  <bool name="suppressHeader">false</bool>
</resources>
```

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:
import android.os.Bundle;
import com.actionbarsherlock.app.SherlockPreferenceActivity;

public class EditPreferences extends SherlockPreferenceActivity {
    @SuppressWarnings("deprecation")
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (Build.VERSION.SDK_INT<Build.VERSION_CODES.HONEYCOMB) {
            addPreferencesFromResource(R.xml.preferences);
        }
    }

    @TargetApi(Build.VERSION_CODES.HONEYCOMB)
    @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:

private void editPrefs() {
    Intent i=new Intent(this, EditPreferences.class);
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB 
        && getResources().getBoolean(R.bool.suppressHeader)) {
        i.putExtra(PreferenceActivity.EXTRA_NO_HEADERS, true);
        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);
}
If we are on API Level 10 or below, where we do not have preference fragments, we start up EditPreferences as before. If `R.bool.suppressHeaders` is false – as it will be on `-large` and `-xlarge` screens — we also start up EditPreferences as before. But, if `R.bool.suppressHeaders` is true, then we will add three extras to our Intent:

- **EXTRA_NO_HEADERS**, set to true, to indicate that we do not want the headers to be displayed
- **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 three 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.
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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book’s GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.
Step #1: Adding a StockPreferenceFragment

In the preceding chapter, we saw StockPreferenceFragment, which simply loads a <PreferenceScreen> bit of XML for us. This is simpler than rolling our own custom PreferenceFragment implementations, so let's use it.

If you wish to make this change using Eclipse's structured resource editor, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

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 StockPreferenceFragment in the “Name” field. Click the “Browse...” button next to the “Superclass” field and find PreferenceFragment to set as the superclass. Then, click “Finish” on the new-class dialog to create the StockPreferenceFragment class.

Then, with StockPreferenceFragment open in the editor, paste in the following class definition:

```java
package com.commonsware.empublite;

import android.annotation.TargetApi;
import android.os.Bundle;
import android.preference.PreferenceFragment;

@TargetApi(11)
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);
    }
}
```
Outside of Eclipse

Create a `src/com/commonsware/empublite/StockPreferenceFragment.java` source file, with the content shown above.

Step #2: 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). The other will define the preferences that we wish to collect for the one header we will define.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

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, 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">
```

477
Note that while the code listing may show the root element wrapping onto a second line, it really should be all on one line.

Right-click over the xml folder, and choose New > File from the context menu. Fill in pref_display.xml as the name, 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:

```xml
    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.

**Outside of Eclipse**

Add six new `<string>` elements to res/values/strings.xml:

```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>
```
Then, create a res/xml/ directory in your project. In there, create a preference_headers.xml file with the XML from the first code listing in the “Eclipse” section above. Also create a pref_display.xml file with the XML from the second code listing in the “Eclipse” section above.

**Step #3: Creating Our PreferenceActivity**

We now need an implementation of SherlockPreferenceActivity to load our preference XML, using just pref_display.xml on pre-API Level 11 devices and using the full set of XML on API Level 11+ devices. We will use an implementation nearly identical to the one shown in the previous chapter.

If you wish to make this change using Eclipse's wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**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 Preferences in the “Name” field. Click the “Browse...” button next to the “Superclass” field and find SherlockPreferenceActivity to set as the superclass. Then, click “Finish” on the new-class dialog to create the Preferences class.

Then, with Preferences open in the editor, paste in the following class definition:

```java
class Preferences extends SherlockPreferenceActivity {
  @SuppressWarnings("deprecation")
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (Build.VERSION.SDK_INT<Build.VERSION_CODES.HONEYCOMB) {
      addPreferencesFromResource(R.xml.pref_display);
    }
}
```
Eclipse will complain about addPreferencesFromResource() being deprecated – despite the fact that we are only using it on older Android API levels – if we do not have the @SuppressWarnings("deprecation") annotation on the onCreate() method.

Eclipse may also complain that onBuildHeaders() uses loadHeadersFromResource(), which is only available on API Level 11 and higher. That is why we have the @TargetApi(Build.VERSION_CODES.HONEYCOMB) annotation attached to onBuildHeaders(), to indicate that we are aware of this. In this case, since onBuildHeaders() is only ever called on API Level 11 or higher, there is no need to check Build.VERSION.SDK_INT to confirm this fact.

Then, open up AndroidManifest.xml in Eclipse and switch to the “Application” sub-tab. Scroll down to the “Application Nodes” list and click the “Add...” button, choosing to add a new activity. Click the “Browse...” button next to “Name” and pick the Preferences class. Then save your changes (e.g., <Ctrl>-<S>).

**Outside of Eclipse**

Create a src/com/commonsware/empublite/Preferences.java source file, with the content shown above.

Also, add the following element as a child of the <application> element in your AndroidManifest.xml file:

```
<activity android:name="Preferences">
</activity>
```

**Step #4: 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.
If you wish to make this change using Eclipse's wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**Eclipse**

Double-click on the `res/menu/options.xml` file in your project. Click the “Add...” button to add a new menu item. Give it the following details:

- **Id of @id/settings**
- **Title of @string/settings** (using the “Browse...” button to define a new string, with a value of Settings)
- **Icon of @android:drawable/ic_menu_preferences**
- “Show as action” of never

![Attributes for Item1 (Item)](image)

*Figure 182: Settings Menu Item*

Use the “Up” and “Down” buttons to move this new menu item to be the first one in the list.
TUTORIAL #13 - USING SOME PREFERENCES

Outside of Eclipse
Add the following XML element to res/menu/options.xml as the first child of the
<menu> root element:
<item
android:id="@+id/settings"
android:icon="@android:drawable/ic_menu_preferences"
android:showAsAction="never"
android:title="@string/settings">
>
</item>

You will also need to add a settings string resource, with a value of Settings.

Step #5: 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
new Intent(this
this, Preferences.class));
return
return(true
true);

Now, if you run this in an emulator or device, you will see the new option in the
action bar overflow:

482

Subscribe to updates at https://commonsware.com

Special Creative Commons BY-NC-SA 4.0 License Edition


Choosing the “Settings” option brings up the list of preference headings:
Tapping on the “Display and Navigation” heading brings up our two preferences:

Figure 184: Our Preference Headings
On a tablet, we see the headings and the selected headings' worth of preferences at the same time:

Figure 185: Our Preferences
Step #6: Loading Our Preferences

Collecting those preferences is one thing. Actually *using* them requires yet more work.

Our first step is to load our SharedPreferences object. This will read the persisted preferences and make them available to us for examination (and, as we will see, modification). Any changes made to those preferences — say, from the Preferences activity — will be automatically reflected in the loaded SharedPreferences.

However, since the persisted preferences are *persisted* — meaning that they are stored in a file — we need to try to load them in the background. Our ModelFragment already has some load-the-data-in-the-background logic, so we can extend that to set up the SharedPreferences.

Open up ModelFragment and add two more data members to the class:

```java
private SharedPreferences prefs = null;
private PrefsLoadTask prefsTask = null;
```
UPDATE  #13 - USING SOME PREFERENCES

We will need to define that PrefsLoadTask as follows:

```java
private class PrefsLoadTask extends AsyncTask<Context, Void, Void> {
    SharedPreferences localPrefs = null;

    @Override
    protected Void doInBackground(Context... ctxt) {
        localPrefs = PreferenceManager.getDefaultSharedPreferences(ctxt[0]);
        localPrefs.getAll();
        return(null);
    }

    @Override
    public void onPostExecute(Void arg0) {
        ModelFragment.this.prefs = localPrefs;
        ModelFragment.this.prefsTask = null;
        deliverModel();
    }
}
```

Here, we call getDefaultSharedPreferences() in doInBackground(). We also call getAll() on the SharedPreferences object, to make sure that it is fully loaded from disk, in case Android has an optimization that lazy-loads the preference data on first use. In onPostExecute(), we store the resulting SharedPreferences in a data member, clear our prefsTask data member (indicating that we are done with the load), and call deliverModel().

The deliverModel() method will also need to be adjusted, to hand over the SharedPreferences to the EmPubLiteActivity:

```java
synchronized private void deliverModel() {
    if (prefs != null && contents != null) {
        ((EmPubLiteActivity) getActivity()).setupPager(prefs, contents);
    } else {
        if (prefs == null && prefsTask == null) {
            prefsTask = new PrefsLoadTask();
            executeAsyncTask(prefsTask, getActivity().getApplicationContext());
        }
        if (contents == null && contentsTask == null) {
            contentsTask = new ContentsLoadTask();
            executeAsyncTask(contentsTask, getActivity().getApplicationContext());
        }
    }
}
```
Here, we initialize either or both of our tasks if we do not have our data (e.g., when `deliverModel()` is first called), and we only pass the data to the activity when we have both the BookContents and the SharedPreferences.

Of course, `setupPager()` in `EmPubLiteActivity` needs to be updated to match:

```java
void setupPager(SharedPreferences prefs, BookContents contents) {
    this.prefs = prefs;
    adapter = new ContentsAdapter(this, contents);
    pager.setAdapter(adapter);

    findViewById(R.id.progressBar1).setVisibility(View.GONE);
    findViewById(R.id.pager).setVisibility(View.VISIBLE);
}
```

This will require a `SharedPreferences` data member to be added to `EmPubLiteActivity` as well:

```java
private SharedPreferences prefs = null;
```

**Step #7: Saving the Last-Read Position**

The 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`:

```java
private static final String PREF_LAST_POSITION = "lastPosition";
```

Then, add the following implementation of `onPause()` to `EmPubLiteActivity`:

```java
@Override
public void onPause() {
    if (prefs != null) {
        int position = pager.getCurrentItem();
        prefs.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. Since this project has API Level 9 as the minimum SDK version, it is safe for us to use apply() instead of the older synchronous commit().

**Step #8: 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:

```java
private static final String PREF_SAVE_LAST_POSITION = "saveLastPosition";
```

Add the following lines to setupPager() in EmPubLiteActivity:

```java
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:

```java
private static final String PREF_KEEP_SCREEN_ON = "keepScreenOn";
```

Then, add one more line to setupPager() in EmPubLiteActivity:

```java
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 override onResume() to also update the screen-on setting:

```java
@Override
public void onResume() {
    super.onResume();
    if (prefs != null) {
        pager.setKeepScreenOn(prefs.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.
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 and how to use it in environments other than Android, a fine book is *The Definitive Guide to SQLite* by Michael Owens.
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:
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/Constants sample application. This application pulls a bunch of gravitational constants from the SensorManager class, puts them in a database table, displays them in a SherlockListFragment, 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):

```java
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:

```java
@Override
public void onCreate(SQLiteDatabase db) {
```
```
    db.execSQL("CREATE TABLE constants (_id INTEGER PRIMARY KEY AUTOINCREMENT,
            title TEXT, value REAL);");

    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");
```
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:

```java
@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()`.
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 getWriteableDatabase(), depending upon whether or not you will be changing its contents.

For example, the ConstantsFragment from the sample app creates a DatabaseHelper instance in onActivityCreated() and holds onto it in a data member:

```java
public class ConstantsFragment extends SherlockListFragment implements DialogInterface.OnClickListener {
    private DatabaseHelper db = null;

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        setHasOptionsMenu(true);
        setRetainInstance(true);

        db = new DatabaseHelper(getActivity());
        new LoadCursorTask().execute();
    }
}
```

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():

```java
@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 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:

```java
db = new DatabaseHelper(getActivity());
```

in a fragment, with db as a data member, you might have:

```java
db = new DatabaseHelper(getActivity().getApplicationContext());
```

with db as a static data member, shared by multiple activities or other components. We will examine this pattern in greater detail later in this book.

**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().

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.

For example, ConstantsFragment has a doQuery() method that uses rawQuery():

```java
private Cursor doQuery() {
    String query =
        String.format("SELECT _id, %s, %s FROM %s ORDER BY %s",
            DatabaseHelper.TITLE, DatabaseHelper.VALUE,
            DatabaseHelper.TABLE, DatabaseHelper.TITLE);

    return (db.getReadableDatabase().rawQuery(query, null));
}
```

**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.

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:

```java
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();
```

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:

```java
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
@SuppressWarnings("deprecation")
@Override
public void onPostExecute(Void arg0) {
    SimpleCursorAdapter adapter;

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        adapter =
            new SimpleCursorAdapter(
                getActivity(),
```
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
<?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>
```
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).

We use two different versions of the SimpleCursorAdapter constructor because one was deprecated in API Level 11. We use the Build class to detect which API level we are on and choose the right constructor accordingly. We will go into this technique in greater detail in a later chapter.

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, ...). We will see one workaround for this later in this chapter.

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:

```java
@Override
public void onDestroy() {
    super.onDestroy();

    ((CursorAdapter)getListAdapter()).getCursor().close();
    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 onActivityCreated() (shown above). 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):
private class LoadCursorTask extends AsyncTask<Void, Void, Void> {
    private Cursor constantsCursor = null;

    @Override
    protected Void doInBackground(Void... params) {
        constantsCursor = doQuery();
        constantsCursor.getCount();
        return (null);
    }

    @TargetApi(Build.VERSION_CODES.HONEYCOMB)
    @SuppressWarnings("deprecation")
    @Override
    public void onPostExecute(Void arg0) {
        SimpleCursorAdapter adapter =
            new SimpleCursorAdapter(
                get.Activity(),
                R.layout.row,
                constantsCursor,
                new String[] { DatabaseHelper.TITLE,
                                DatabaseHelper.VALUE },
                new int[] { R.id.title, R.id.value },
                0);

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
            adapter =
                new SimpleCursorAdapter(
                    get.Activity(),
                    R.layout.row,
                    constantsCursor,
                    new String[] { DatabaseHelper.TITLE,
                                   DatabaseHelper.VALUE },
                    new int[] { R.id.title, R.id.value });
        }
        else {
            adapter =
                new SimpleCursorAdapter(
                    get.Activity(),
                    R.layout.row,
                    constantsCursor,
                    new String[] { DatabaseHelper.TITLE,
                                   DatabaseHelper.VALUE },
                    new int[] { R.id.title, R.id.value });
        }
        setListAdapter(adapter);
    }
}

We execute the actual query in doInBackground(), holding onto it in a data member of the LoadCursorTask. We also call getCount() on the Cursor, to force it to actually perform the query — rawQuery() returns the Cursor, but the query is not actually executed until we do something that needs the result set.

onPostExecute() then wraps it in a SimpleCursorAdapter and attaches it to the ListView via setListAdapter() on our SherlockListFragment.
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 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 data definition language (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:

```java
    db.execSQL("CREATE TABLE constants (_id INTEGER PRIMARY KEY AUTOINCREMENT,
                   title TEXT, value REAL);");
```

This will create a table, named constants, with a primary key column named _id that is an auto-incremented integer (i.e., SQLite will assign the value for you when you insert rows), plus two data columns: title (text) and value (a float, or “real” in SQLite terms). SQLite will automatically create an index for you on your primary key column — you could add other indexes here via some CREATE INDEX statements, if you wanted.
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:

```java
    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.

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:

![Figure 187: The ConstantsBrowser Sample](image)

Clicking on that brings up a dialog — a technique we will discuss later in this book:
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:

```java
private class InsertTask extends AsyncTask<ContentValues, Void, Void> {
    private Cursor constantsCursor = null;

    @Override
    protected Void doInBackground(ContentValues... values) {
        db.getWritableDatabase().insert(DatabaseHelper.TABLE, DatabaseHelper.TITLE, values[0]);

        constantsCursor = doQuery();
        constantsCursor.getCount();

        return(null);
    }

    @Override
    public void onPostExecute(Void arg0) {
        ((CursorAdapter)getListAdapter()).changeCursor(constantsCursor);
    }
}
```

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.

Then, in onPostExecute(), we can safely update the UI with the new Cursor. We do this by calling changeCursor() on our CursorAdapter, retrieved from the fragment via getListAdapter(). changeCursor() will swap out our old Cursor in our SimpleCursorAdapter with the new one, automatically updating the ListView along the way.

**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:

```java
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.

**Leveraging ROWID**

The _id requirement of CursorAdapter is annoying. However, SQLite provides us with a workaround: ROWID.

Quoting the SQLite documentation:

In SQLite, every row of every table has a 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.

The Database/ConstantsROWID sample application demonstrates this. It is a clone of the Database/Constants project shown earlier in this chapter, but it skips its own _id column and uses ROWID instead.

There are only two changes of significance. First, DatabaseHelper no longer has _id in the schema used with execSQL() in onCreate():

```java
db.execSQL("CREATE TABLE constants (title TEXT, value REAL);");
```
Second, the doQuery() method in ConstantsFragment queries on ROWID, but uses the SQL AS keyword to rename it as _id:

```java
private Cursor doQuery() {
    String query =
        String.format("SELECT ROWID AS _id, %s, %s FROM %s ORDER BY %s",
                      DatabaseHelper.TITLE,
                      DatabaseHelper.VALUE,
                      DatabaseHelper.TABLE,
                      DatabaseHelper.TITLE);

    return db.getReadableDatabase().rawQuery(query, null);
}
```

Since nothing else in the project used the old _id value, nothing else needs to change.

**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.
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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book’s GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.

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.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).
Eclipse

Open res/values/strings.xml and add a new string resource, named on_upgrade_error, with a value of This should not be called.

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 DatabaseHelper in the “Name” field. Click the “Browse...” button next to the “Superclass” field and find 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 following class definition:

```java
package com.commonsware.empublite;

import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.os.AsyncTask;

public class DatabaseHelper extends SQLiteOpenHelper {
  private static String DATABASE_NAME = "empublite.db";
  private static int SCHEMA_VERSION = 1;
  private static DatabaseHelper singleton = null;
  private Context ctxt = 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);
    this.ctxt = ctxt;
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    try {
      db.beginTransaction();
      db.execSQL("CREATE TABLE notes (position INTEGER PRIMARY KEY, prose TEXT);");
      db.setTransactionSuccessful();
    } finally {
```

512
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    throw new RuntimeException(ctx.getString(R.string.on_upgrade_error));
}

Outside of Eclipse

Create a src/com/commonsware/empublite/DatabaseHelper.java source file, with the content shown above. Also, add a new string resource, named on_upgrade_error, with a value of This should not be called.

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). It also holds onto the supplied Context for use later in this chapter. 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). We wrap this in our own transaction for illustration purposes, though in this case, since there is only one SQL statement, it is not strictly necessary.
- 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.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**Eclipse**

Right click over the res/layout/ directory in your project, and choose New > File from the context menu. Give the file a name of editor.xml. Then, 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.

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 SherlockFragment 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 following class definition:

```java
package com.commonsware.empublite;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;
import com.actionbarsherlock.app.SherlockFragment;
import com.actionbarsherlock.view.Menu;
import com.actionbarsherlock.view.MenuItem;
```

514
public class NoteFragment extends SherlockFragment {
    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);
        int position = getArguments().getInt(KEY_POSITION, -1);

        editor = (EditText) result.findViewById(R.id.editor);

        return result;
    }
}

Outside of Eclipse

Create a src/com/commonsware/empublite/NoteFragment.java source file, with the content shown in the code listing in the “Eclipse” section above.

Then, create a res/layout/editor.xml file with the following XML:

```xml
<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:hint="@string/hint"
    android:inputType="textMultiLine"/>
```

You will also need to add a new <string> element in your res/values/strings.xml file, with a name of hint and a value like Enter notes here.
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.

If you wish to make this change using Eclipse's wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

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 NoteActivity in the “Name” field. Click the “Browse...” button next to the “Superclass” field and find SherlockFragmentActivity to set as the superclass. Then, click “Finish” on the new-class dialog to create the NoteActivity class.

Then, with NoteActivity open in the editor, paste in the following class definition:

```java
package com.commonsware.empublite;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import com.actionbarsherlock.app.SherlockFragmentActivity;
```
public class NoteActivity extends SherlockFragmentActivity {
    public static final String EXTRA_POSITION="position";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getSupportFragmentManager().findFragmentById(android.R.id.content)==null) {
            int position=getIntent().getIntExtra(EXTRA_POSITION, -1);
            if (position>=0) {
                Fragment f=NoteFragment.newInstance(position);
                getSupportFragmentManager().beginTransaction()
                    .add(android.R.id.content, f).commit();
            }
        }
    }
}

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.

You will also need to add a new activity node to the list of nodes in the Application sub-tab of AndroidManifest.xml, pointing to NoteActivity, following the same approach that we used for other activities in this application.

**Outside of Eclipse**

Create a src/com/commonsware/empublite/NoteActivity.java source file, with the content shown in the code listing in the “Eclipse” section above. Also add the corresponding <activity> element in the manifest:

```
<activity android:name="NoteActivity"/>
```
Step #6: Loading and Saving Notes

So, we have a database, a fragment, and an activity. If we started up the activity, the user could type in some notes... which would not be stored, nor loaded, from the database. Hence, we have a bit more work to do before we let the user into this UI.

In addition, since database I/O can be slow, we really need to ensure that we are loading and saving our notes asynchronously. In particular, we need to allow the NoteFragment to load a note, yet get that note's prose back via some sort of asynchronous mechanism, rather than having some sort of blocking call.

In this tutorial, we will isolate much of the database-handling logic on the DatabaseHelper, beyond what is required by the SQLiteOpenHelper abstract class.

To that end, let us first set up an interface, NoteListener, defined as an inner interface of DatabaseHelper:

```java
interface NoteListener {
    void setNote(String note);
}
```

For the asynchronous work, we can use our good friend AsyncTask. First, let us load a note from the database, by defining a GetNoteTask as an inner class of our DatabaseHelper:

```java
private class GetNoteTask extends AsyncTask<Integer, Void, String> {
    private NoteListener listener = null;

    GetNoteTask(NoteListener listener) {
        this.listener = listener;
    }

    @Override
    protected String doInBackground(Integer... params) {
        String[] args = { params[0].toString() }; // 1

        Cursor c = getReadableDatabase().rawQuery("SELECT prose FROM notes WHERE position=?", args);
        c.moveToFirst(); // 2

        if (c.isAfterLast()) { // 3
            return(null);
        }
    }
}
```
String result=c.getString(0);
c.close();
return(result);
}

@Override
public void onPostExecute(String prose){
listener.setNote(prose);
}

This is a regular inner class, not a static inner class, which should cause you to pause for a moment — a regular inner class can be dangerous with configuration changes. However, in this case, this is an inner class of our DatabaseHelper, which is a singleton and will be unaffected directly by any configuration changes.

GetNoteTask will need two pieces of data:

• the position (i.e., chapter) whose notes we need to load, which will be supplied as an Integer to doInBackground() by way of execute(), and
• a NoteListener that we can supply the loaded prose to, which GetNoteTask takes in its constructor

Our doInBackground() gets a readable database from DatabaseHelper and proceeds to use rawQuery() to retrieve the prose, given the position, returning null if there is no such note (e.g., the user is trying to edit the note for a chapter for the first time). The prose is returned by doInBackground() and supplied to onPostExecute(), which turns around and calls setNote() on our NoteListener to pass it along to the UI on the main application thread.

Similarly, we will need a SaveNoteTask as an inner class of DatabaseHelper:

private class SaveNoteTask extends AsyncTask<Void, Void, Void> {
private int position;
private String note=null;

SaveNoteTask(int position, String note) {
this.position=position;
this.note=note;
}

@Override
protected Void doInBackground(Void... params){
String[] args= { String.valueOf(position), note };
In this case, we need both the position and the note to be saved, which SaveNoteTask collects in its constructor. In doInBackground(), we use SQLite’s INSERT OR REPLACE SQL statement to either INSERT a new note or UPDATE an existing note, based on the supplied position. We could avoid this by tracking whether or not we had a note from our GetNoteTask and using the insert() and update() methods on SQLiteDatabase, but the INSERT OR REPLACE approach is a bit more concise in this case.

To invoke those tasks, we can create methods on DatabaseHelper:

```java
void getNoteAsync(int position, NoteListener listener) {
    ModelFragment.executeAsyncTask(new GetNoteTask(listener), position);
}

void saveNoteAsync(int position, String note) {
    ModelFragment.executeAsyncTask(new SaveNoteTask(position, note));
}
```

These methods use the static executeAsyncTask() method on ModelFragment that uses execute() or executeOnExecutor() as appropriate.

Over in NoteFragment, we need to use our new getNoteAsync() method to load the note for use in our EditText. To that end, add the following statement to the onCreateView() in NoteFragment, just before the return:

```java
DatabaseHelper.getInstance(getActivity()).getNoteAsync(position, this);
```

Here, we retrieve our singleton DatabaseHelper and tell it to load our note, passing the results to ourself as the NoteListener.

For that to work, though, we will need to add implements DatabaseHelper.NoteListener to the class declaration:

```java
public class NoteFragment extends SherlockFragment implements DatabaseHelper.NoteListener {

    That, in turn, requires us to implement setNote():
```
### Step #7: Add Notes to the Action Bar

Now, we can let our user actually start working with the notes, by giving them 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:

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/notes"
    android:icon="@android:drawable/ic_menu_edit"
    android:showAsAction="ifRoom|withText"
    android:title="@string/notes">
  </item>
  <item
    android:id="@+id/settings"
    android:icon="@android:drawable/ic_menu_preferences"
    android:showAsAction="never"
    android:title="@string/settings">
  </item>
</menu>
```
Eclipse users 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():

```java
    case R.id.notes:
        Intent i = new Intent(this, NoteActivity.class);
        i.putExtra(NoteActivity.EXTRA_POSITION, pager.getCurrentItem());
        startActivity(i);
        return true;
```

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.

If you build and run the app on a device or emulator, you will see the new toolbar button in the action bar:
Figure 189: The New Action Bar Item

Tapping that will bring up the notes for whatever chapter 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 chapter.

Step #8: Support Deleting Notes

So, we can now add and edit notes. However, the only way we can “delete” a note is to blank out the EditText. While that works, it would be nice to offer a cleaner delete option.

First, we need to add some more logic to DatabaseHelper to delete notes. As with getting and saving notes, this will involve an AsyncTask and a method to execute an instance of that task.

With that in mind, add the following inner class to DatabaseHelper:

```java
private class DeleteNoteTask extends AsyncTask<Integer, Void, Void> {
    @Override
```
Here, given the position (supplied as an Integer to doInBackground()), we execute a DELETE statement to get rid of it. This could also have been implemented using the delete() method on SQLiteDatabase.

Then, add a deleteNoteAsync() method to DatabaseHelper, to invoke our DeleteNoteTask:

```java
void deleteNoteAsync(int position) {
    ModelFragment.executeAsyncTask(new DeleteNoteTask(), position);
}
```

Our full DatabaseHelper at this point should look like:

```java
package com.commonsware.empublite;

import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.os.AsyncTask;

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;
    private Context ctxt = 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);
        thisctxt = ctxt;
    }

    @Override
}
```
public void onCreate(SQLiteDatabase db) {
    try {
        db.beginTransaction();
        db.execSQL("CREATE TABLE notes (position INTEGER PRIMARY KEY, prose TEXT);");
        db.setTransactionSuccessful();
    } finally {
        db.endTransaction();
    }
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    throw new RuntimeException(
        ctxt.getString(R.string.on_upgrade_error));
}

void getNoteAsync(int position, NoteListener listener) {
    ModelFragment.executeAsyncTask(new GetNoteTask(listener), position);
}

void saveNoteAsync(int position, String note) {
    ModelFragment.executeAsyncTask(new SaveNoteTask(position, note));
}

void deleteNoteAsync(int position) {
    ModelFragment.executeAsyncTask(new DeleteNoteTask(), position);
}

interface NoteListener {
    void setNote(String note);
}

private class GetNoteTask extends AsyncTask<Integer, Void, String> {
    private NoteListener listener=null;

    GetNoteTask(NoteListener listener) {
        this.listener=listener;
    }

    @Override
    protected String doInBackground(Integer... params) {
        String[] args= { params[0].toString() };

        Cursor c=
            getReadableDatabase().rawQuery("SELECT prose FROM notes WHERE position=?",
                                            args);
        c.moveToFirst();
        if (c.isAfterLast()) {

return(null);
}
String result=c.getString(0);
c.close();
return(result);
}

@Override
public void onPostExecute(String prose) {
    listener.setNote(prose);
}

private class SaveNoteTask extends AsyncTask<Void, Void, Void> {
    private int position;
    private String note=null;

    SaveNoteTask(int position, String note) {
        this.position=position;
        this.note=note;
    }

    @Override
    protected Void doInBackground(Void... params) {
        String[] args= { String.valueOf(position), note };

        getWritableDatabase().execSQL("INSERT OR REPLACE INTO notes (position, prose) VALUES (?, ?)",
                                args);

        return(null);
    }
}

private class DeleteNoteTask extends AsyncTask<Integer, Void, Void> {
    @Override
    protected Void doInBackground(Integer... params) {
        String[] args= { params[0].toString() };

        getWritableDatabase().execSQL("DELETE FROM notes WHERE position=\
                                ?, args);

        return(null);
    }
}

Next, let's create a new resource, res/menu/notes.xml, to configure the action bar for the activity hosting our NoteFragment:
This simply defines a single action bar item, with an ID of delete.

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, use the structured resource editor to add a new menu resource, with an ID of delete, an icon of @android:drawable/ic_menu_delete, and a title that consists of a new delete string resource (with a value of Delete). Also, mark this new item as ifRoom|withText for the “Show as action” item.

To let Android know that our NoteFragment wishes to participate in the action bar, we need to call setHasOptionsMenu(true) at some point. The easiest place to put that would be in our onCreateView() implementation in NoteFragment:

```java
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    View result = inflater.inflate(R.layout.editor, container, false);
    int position = getArguments().getInt(KEY_POSITION, -1);

    editor = (EditText) result.findViewById(R.id.editor);
    DatabaseHelper.getInstance(getActivity()).getBodyAsync(this);

    setHasOptionsMenu(true);

    return result;
}
```

That will trigger a call to onCreateOptionsMenu(), which we will need to add to NoteFragment:

```java
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.notes, menu);

    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:

```java
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.delete) {
        int position = getArguments().getInt(KEY_POSITION, -1);
        isDeleted = true;
        DatabaseHelper.getInstance(getActivity()).deleteNoteAsync(position);
        ((NoteActivity) getActivity()).closeNotes();
        return (true);
    }
    return (super.onOptionsItemSelected(item));
}
```

Here, if the user tapped the delete item, we update an isDeleted data member to track that our note is now deleted, plus use the new deleteNoteAsync() method on DatabaseHelper to actually get rid of the note. We also call a closeNotes() method on our hosting activity to indicate that we are no longer needed on the screen (since the note is deleted).

For this to build, we will need to add isDeleted to NoteFragment as a data member:

```java
private boolean isDeleted = false;
```

The reason for tracking isDeleted is for onPause(), so when our fragment leaves the foreground, we do not inadvertently save it again. So, update onPause() to only do its work if isDeleted is false:

```java
@Override
public void onPause() {
    if (!isDeleted) {
        int position = getArguments().getInt(KEY_POSITION, -1);
        DatabaseHelper.getInstance(getActivity()).saveNoteAsync(position, editor.getText().toString());
    }
    super.onPause();
}
```
The complete NoteFragment, at this point, should look like:

```java
package com.commonsware.empublite;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;
import com.actionbarsherlock.app.SherlockFragment;
import com.actionbarsherlock.view.Menu;
import com.actionbarsherlock.view.MenuItem;

public class NoteFragment extends SherlockFragment implements DatabaseHelper.NoteListener {
    private static final String KEY_POSITION = "position";
    private EditText editor = null;
    private boolean isDeleted = false;

    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);
        int position = getArguments().getInt(KEY_POSITION, -1);
        editor = (EditText) result.findViewById(R.id.editor);
        DatabaseHelper.getInstance(getActivity()).getNoteAsync(position, this);
        setHasOptionsMenu(true);
        return (result);
    }

    @Override
    public void onPause() {
        if (!isDeleted) {
            int position = getArguments().getInt(KEY_POSITION, -1);
            DatabaseHelper.getInstance(getActivity()).saveNoteAsync(position, editor.getText().toString());
        }
    }
}
```
super.onPause();

@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) {
        int position = getArguments().getInt(KEY_POSITION, -1);
        isDeleted = true;
        DatabaseHelper.getInstance(getActivity()).
            deleteNoteAsync(position);

        ((NoteActivity) getActivity()).closeNotes();
        return(true);
    }
    return(super.onOptionsItemSelected(item));
}

@Override
public void setNote(String note) {
    editor.setText(note);
}

However, we also need to implement closeNotes() on NoteActivity, as we are trying to call that from onOptionsItemSelected():

    void closeNotes() {
        finish();
    }

Here, we just call finish() to get rid of the activity and return us to EmPubLiteActivity.

If you run this in a device or emulator, and you go into the notes, you will see our delete toolbar button:
Tapping that toolbar button will delete the note (if there is one) 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.
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.
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.

Introducing the Sample

In this section, we will take a look at the Internet/Weather sample project. This project does several things:

- It requests our location from LocationManager, specifically using GPS
- It retrieves the weather for our location from the US National Weather Service (NWS) for the latitude and longitude we get from LocationManager
- It parses the XML received from the NWS, generates a Web page in response, and displays that Web page in a WebView widget

Later in this book, we will examine the LocationManager portion of this sample. For the moment, we will focus on the Internet access.

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:

```xml
<uses-permission android:name="android.permission.INTERNET"/>
```

A Task for Updating

We have an activity (WeatherDemo) which follows the standard load-the-dynamic-fragment pattern seen throughout this book, this time setting up WeatherFragment:

```java
package com.commonsware.android.weather;

import android.os.Bundle;
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class WeatherDemo extends SherlockFragmentActivity {
    @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 WeatherFragment()).commit();
}

Eventually, when we get a GPS fix, the onLocationChanged() method of WeatherFragment will be called — we will get into the details of how this occurs later in this book when we cover LocationManager. Suffice it to say that it happens, and more importantly, it happens on the main application thread.

We do not want to do Internet access on the main application thread, as we have no idea if it will complete quickly.

So, we set up an AsyncTask, named FetchForecastTask, and execute an instance of it:

```java
@Override
public void onLocationChanged(Location location) {
    FetchForecastTask task = new FetchForecastTask();
    task.execute(location);
}
```

The Location object supplied to onLocationChanged() will contain, among other things, our latitude (getLatitude()) and longitude (getLongitude()) in decimal degrees as Java double values.

The doInBackground() of FetchForecastTask is where we can do all the downloading and parsing of our Web service call:

```java
@Override
protected String doInBackground(Location... locs) {
    String page = null;
    try {
        Location loc = locs[0];
        String url = String.format(template, loc.getLatitude(), loc.getLongitude());
        page = generatePage(buildForecasts(getForecastXML(url)));
    } catch (Exception e) {
```
Internet Access

```java
return (page);
}

@Override
protected void onPostExecute(String page) {
    if (e == null) {
        getWebView().loadDataWithBaseURL(null, page, "text/html",
                                           "UTF-8", null);
    } else {
        Log.e(getClass().getSimpleName(), "Exception fetching data", e);
        Toast.makeText(getActivity(),
                         String.format(R.string.error,
                                       e.toString()), Toast.LENGTH_LONG)
                          .show();
    }
}
}
```

We need to synthesize a URL to access that NWS REST endpoint for getting a forecast based upon latitude and longitude. A template of that URL is held in a string resource named R.string.url (too long to reprint here) and is stored in a data member named template up in onCreate():

```java
template = getActivity().getString(R.string.url);
```

We use the `String.format()` method to pour our latitude and longitude from the Location object into the template to get a fully-qualified URL (fortunately, floating-point numbers do not need to be URL-encoded).

We then execute a series of methods, defined on the fragment, to handle the actual Web service call:

- `getForecastXML()` makes the HTTP request and retrieves the XML from the NWS REST endpoint
- `buildForecasts()` parses that XML into a series of Forecast objects
- `generatePage()` takes the Forecast objects and crafts a Web page to display the forecast

However, these might fail with an Exception (e.g., no connectivity, malformed XML). If one does, we hold onto the Exception in a data member of FetchForecastTask, so we can use it in `onPostExecute()` on the main application thread.
Doing the Internet Thing

The getForecastXML() method uses a fairly typical recipe for fetching data off the Internet from an HTTP URL using HttpURLConnection:

```java
private String getForecastXML(String path) throws IOException {
    BufferedReader reader = null;
    URL url = new URL(path);
    HttpURLConnection c = (HttpURLConnection)url.openConnection();
    try {
        reader = new BufferedReader(new InputStreamReader(c.getInputStream()));
        StringBuilder buf = new StringBuilder();
        String line = null;
        while ((line = reader.readLine()) != null) {
            buf.append(line + "\n");
        }
        return buf.toString();
    } finally {
        if (reader != null) {
            reader.close();
        }
        c.disconnect();
    }
}
```

We create a connection for the URL, then execute the request and use a StringBuilder to get the response back as a String. If this fails for any reason, it will raise an Exception, which will be caught by FetchForecastTask.

Dealing with the Result

The buildForecasts() method uses the DOM to parse the rather bizarre XML format returned by the NWS REST endpoint:

```java
private ArrayList<Forecast> buildForecasts(String raw)
    throws Exception {
    ArrayList<Forecast> forecasts = new ArrayList<Forecast>();
    DocumentBuilderFactory builder =
        DocumentBuilderFactory.newInstance().newDocumentBuilder();
    Document doc = builder.parse(new InputSource(new StringReader(raw)));
    NodeList times = doc.getElementsByTagName("start-valid-time");
    ```
```java
for (int i=0; i < times.getLength(); i++) {
    Element time=(Element)times.item(i);
    Forecast forecast=new Forecast();
    forecasts.add(forecast);
    forecast.setTime(time.getFirstChild().getNodeValue());
}

NodeList temps=doc.getElementsByTagName("value");
for (int i=0; i < temps.getLength(); i++) {
    Element temp=(Element)temps.item(i);
    Forecast forecast=forecasts.get(i);
    forecast.setTemp(Integer.valueOf(temp.getFirstChild().getNodeValue()));
}

NodeList icons=doc.getElementsByTagName("icon-link");
for (int i=0; i < icons.getLength(); i++) {
    Element icon=(Element)icons.item(i);
    Forecast forecast=forecasts.get(i);
    forecast.setIcon(icon.getFirstChild().getNodeValue());
}
return(forecasts);
```

(using SAX might be faster in this case — the proof of this is left as an exercise to the reader)

That XML is converted into a series of Forecast objects, representing the triple of time, projected temperature, and a code identifying the projected weather. That code maps to a series of icons up on the NWS Web site.

The generatePage() method takes those Forecast objects and generates a trivial HTML page with a table containing the results.

Back in our FetchForecastTask, onPostExecute() loads that HTML into a WebView via loadDataWithBaseUrl(). The WebView comes from our parent class, WebViewFragment, a port of the native WebViewFragment from Android that works on Android 2.x and ActionBarSherlock, as the native WebViewFragment only exists on API Level 11 and higher:

```java
import android.os.Build;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
```
import android.view.ViewGroup;
import android.webkit.WebView;
import com.actionbarsherlock.app.SherlockFragment;

/**
 * A fragment that displays a WebView.
 * <p>
 * The WebView is automatically paused or resumed when the
 * Fragment is paused or resumed.
 */
public class WebViewFragment extends SherlockFragment {

    private WebView mWebView;
    private boolean mIsWebViewAvailable;

    public WebViewFragment() {
    }

    /**
     * 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(getActivity());
        mIsWebViewAvailable = true;
        return mWebView;
    }

    /**
     * Called when the fragment is visible to the user and
     * actively running. Resumes the WebView.
     */
    @TargetApi(11)
    @Override
    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;
}

(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)

If we encountered an Exception, though, onPostExecute() logs the stack trace to
LogCat, plus displays a Toast to let the user know of our difficulty.)
Running the Sample

When you run the sample, initially it will appear as though the forecast is heavy fog, or perhaps a blizzard:

![Figure 191: The Weather Demo, As Initially Launched](image)

That is because Android is waiting on a GPS fix. If you are running on an actual piece of Android hardware with an enabled GPS receiver (and you live within the area covered by the US National Weather Service's REST API), you should get a forecast shortly.

If you are testing on an emulator, you can fake a GPS fix via DDMS. In Eclipse, go to the DDMS perspective, click on your emulator in the Devices tool, then click on the Emulator Control tool and scroll down to the Location Controls:
INTERNET ACCESS

Here, you can fill in a longitude and latitude (pay attention to the order!), then click “Send”. The fields are filled in by default with the location of the Google headquarters in Mountain View, CA, so simply clicking “Send” will bring up the weather forecast for the Googleplex:

Figure 192: The DDMS Perspective, with Location Controls

542
INTERNET ACCESS

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.

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](https://commonsware.com). 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 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 the next chapter.

**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 [JT itr](twitter) to access Twitter’s API
- Using [Amazon’s JAR](http://www.amazon.com) to access various AWS APIs, including S3, SimpleDB, and SQS
- Using the [Dropbox SDK](http://dropbox.com) 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.

**SSL**

Of course, if you are thinking about HTTP, you really should be thinking about HTTPS — SSL-encrypted HTTP operations.
INTERNET ACCESS

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 `Twitter` for accessing Twitter’s API. 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 three libraries, published by Square, that exemplify this approach: OkHTTP, Retrofit, and Picasso. 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 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.

Using OkHTTP is mostly a matter of dropping its JAR into your project’s libs/ directory, 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.

Also, if you had been using Apache’s HttpClient, and you wanted to stick with that API, there is an optional okhttp-apache that offers an HttpClient-compatible API, but one that uses the OkHTTP engine under the covers.

As a simple incremental improvement over classic HTTP client code, OkHTTP is fairly easy to use. 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.

Note that OkHTTP is used as the standard implementation of HttpURLConnection in Android 4.4 and higher.

**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 has a single activity, with a single fragment, that will download the latest 30 questions in the android tag from StackOverflow, displaying them in a ListView. Tapping on an entry in the list will bring up the user’s default Web browser on that particular question.
Note that StackOverflow 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.

Note that the sample app uses native fragments, and therefore will only run on API Level 11 and higher devices. It should be possible to backport this to use the Android Support package's edition of fragments — the proof of this is left as an exercise to the reader.

**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 500KB, 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 Data Model**

You need to create Java classes that mirror the JSON that you wish to get back from HTTP operations performed against the targeted Web service.

In our case, we are going to use a specific endpoint of the StackExchange API, referred to as /questions after the distinguishing portion of the path. The documentation for this endpoint can be found in the StackExchange 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):

```json
{  
  "items": [    
  
    "question_id": 17196927,    
    "creation_date": 1371660594,    
    "last_activity_date": 1371660594,

```
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.

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:

```java
package com.commonsware.android.retrofit;

public class Item {
  String title;
}```
Internet Access

```java
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 StackExchange. 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...):

```java
package com.commonsware.android.retrofit;

import java.util.List;

public class SOQuestions {
    List<Item> items;
}
```

Having an `items` data member that is a `List` of `Item` tells Retrofit 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.

Creating Your Service Interface

The next thing we need is to tell Retrofit more about where this 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
- 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 StackExchange's API to get questions from StackOverflow:
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 StackExchange 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 StackExchange 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 question() 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:

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:

```java
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

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:
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:

```java
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:

```java
public void success(SOQuestions questions, Response response) {
}
```

We will take a look at the success() implementation shortly.

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:

```java
@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);
}
```

**The Rest of the Story**

Our app’s manifest is unremarkable, other than requesting the INTERNET permission via `<uses-permission>`.
On the surface, our main activity (MainActivity) also seems boilerplate, with a standard onCreate() implementation to load a QuestionsFragment dynamically into the main content area:

```java
package com.commonsware.android.retrofit;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;

public class MainActivity extends Activity implements QuestionsFragment.Contract {
    @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();
        }
    }

    @Override
    public void showItem(Item item) {
        startActivity(new Intent(Intent.ACTION_VIEW, Uri.parse(item.link)));
    }
}
```

However, it implements an interface named QuestionsFragment.Contract, supplied by QuestionsFragment. That interface requires an implementation of a showItem() method. That method will be called when the user indicates that she wishes to view one of the questions shown by the QuestionsFragment. In our case, we just launch an activity to view the Web page indicated by the Item link data member, populated by Retrofit from the StackExchange REST API.

That Contract interface is defined as an inner interface of the QuestionsFragment class:

```java
interface Contract {
    void showItem(Item item);
}
```

This is the contract pattern, espoused by Jake Wharton of ActionBarSherlock fame. 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:
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.

QuestionsFragment itself extends from a ContractListFragment, tailoring it to its own Contract interface:
QuestionsFragment is also our implementation of the Retrofit Callback interface, as was described earlier in this section.

The `onCreateView()` of QuestionsFragment lets the superclass handle the UI setup, but also:

- Marks this as a retained fragment, as we are performing asynchronous operations and want a stable platform for those background threads to communicate back to our UI layer
- Kicks off the request to retrieve the StackOverflow questions, using Retrofit

```java
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    View result = super.onCreateView(inflater, container, savedInstanceState);
    setRetainInstance(true);

    RestAdapter restAdapter = new RestAdapter.Builder().setEndpoint("https://api.stackexchange.com").build();
    StackOverflowInterface so = restAdapter.create(StackOverflowInterface.class);
    so.questions("android", this);
    return result;
}
```

The `success()` Callback method calls `setListAdapter()`, wrapping the items array in a simple `ItemsAdapter`:

```java
class ItemsAdapter extends ArrayAdapter<Item> {
    ItemsAdapter(List<Item> items) {
        super(getActivity(), android.R.layout.simple_list_item_1, items);
    }
}
```

```java
@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));
}
```
The ItemsAdapter uses Html.fromHtml() to populate the ListView rows, not because StackOverflow hands back titles with HTML tags, but because StackOverflow hands back titles with HTML entity references, and Html.fromHtml() should handle many of those.

In addition to the failure() implementation shown previously, the only remaining method is onListItemClick(), where we retrieve the clicked-upon Item and pass that to the Contract via showItem(), to cause the question to be shown to the user:

```java
@Override
cpublic void onListItemClick(ListView l, View v, int position, long id) {
    getContract().showItem(((ItemsAdapter) getListAdapter()).getItem(position));
}
```

The net result is that when we run our application, we see a list of recently-asked Android questions on StackOverflow:

![Figure 194: The Retrofit Demo App](image)
**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, StackOverflow 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](https://commonsware.com) 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.

**Downloading and Installing Picasso**

Picasso can be downloaded as a small JAR from the aforementioned Web site. It has no hard dependencies. However, if OkHTTP happens to be available, Picasso will use it, when you request to load the image from a Web server.
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 StackExchange 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”):

```java
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 StackExchange 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:

```java
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:

```java
Picasso.with(getActivity()).load(item.owner.profileImage)
    .resize(size, size).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 profile_image data member of the Owner inside an Item referred to as item
- Say that we want to resize() the image to a particular size (more on this in a bit)
- 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)
- 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`:

```java
@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)
        .resize(size, size).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 size that we are using comes from our new `ItemsAdapter` constructor:

```java
ItemsAdapter(List<Item> items) {
    super(getActivity(), R.layout.row, R.id.title, items);

    size = getActivity().getResources()
        .getDimensionPixelSize(R.dimen.icon);
}
```

The `ImageView` is sized to be 48dp square. The images coming from StackExchange presently are 32x32 pixels. We are having Picasso handle the resizing for us, so we need to indicate how many pixels per side we want the image to be. To know that, we need to convert 48dp to pixels.

The 48dp value is held in a dimension resource (`R.dimen.icon`). The row layout uses that dimension resource for the `ImageView` height and width. In the constructor, we...
also retrieve that dimension value, converted by Android to pixels based on the actual screen density. The size data member is initialized in the constructor for performance, as the size will be the same for all rows, so there is no sense in going through the calculations for each individual row.

The result is that we now have icons next to each of our question titles:

![Figure 195: The Picasso Demo App](image)

**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:

```java
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    View result = super.onCreateView(inflater, container, savedInstanceState);
    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():

```java
@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 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:

```java
@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")
             .getAsJsonObject());
    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
- `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. There is no support mechanism, except perhaps via ad-hoc social media inquiries and general support sites (e.g., StackOverflow).

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 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.
Intents, Intent Filters, Broadcasts, and Broadcast Receivers

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, plus another channel of the Intent message bus: the broadcast Intent.

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
**INTENTS, INTENT FILTERS, BROADCASTS, AND BROADCAST RECEIVERS**

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).

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 or ALTERNATIVE categories.
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:

- 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:

```xml
<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:

```java
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:

```xml
<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.
**INTENTS, INTENT FILTERS, BROADCASTS, AND BROADCAST RECEIVERS**

FauxSender can retrieve those extras and make use of them:

```java
package com.commonsware.android.fsender;

import android.app.Activity;
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:
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:

```java
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.
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:
The user can tap on either of those two items in the list to have that particular activity be the one to process this event. And, if the user checks the “Use by default for this action” checkbox, 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 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 check that checkbox, 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 the default checkbox is usually inappropriate. Just because the user on Monday chose to send something via Bluetooth and accidentally checked that checkbox 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.
You can elect to force a chooser to display, regardless of the state of that checkbox. 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:

```java
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 198: Your Tailored Chooser Dialog](image)

Notice the lack of the default checkbox — not only must the user make a choice now, but also they cannot make a default choice for the future, either.
Broadcasts and 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, 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.

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:

```java
new IntentFilter(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.

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 onReceive() itself processes, such as forking a thread.

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.
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.

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:
OnBootReceiver simply logs a message to LogCat:

```java
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. 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.

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:
This layout is used by a BatteryFragment, which registers to receive the ACTION_BATTERY_CHANGED Intent in onResume() and unregisters in onPause():

```java
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;
import com.actionbarsherlock.app.SherlockFragment;

public class BatteryFragment extends SherlockFragment {
    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);

        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:
        }
    }
};
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)
- 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:
Sticky Intents and the Battery

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.
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.

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).

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 it yourself. However, it does present a few challenges. In this section, we will examine the Internet/Download sample project, one that uses DownloadManager.
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
<?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:smallScreens="true"
        android:normalScreens="true"
        android:largeScreens="true"
        android:anyDensity="true"/>

    <uses-sdk
        android:minSdkVersion="9"
        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="@style/Theme.Sherlock">
        <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>
```

Note that the manifest also has android:minSdkVersion="9", because that was the API level in which the DownloadManager was introduced.
The Layout

Our sample application has a simple layout, consisting of three buttons:

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
<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="0dp"
            android:layout_weight="1"
            />
    <Button android:id="@+id/query"
            android:text="@string/query_status"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            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="0dp"
            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`:

```java
@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:

```java
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")
```

588
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:

```java
private void queryStatus(View v) {
    Cursor c =
```

```java
mgr.query(new DownloadManager.Query().setFilterById(lastDownload));

if (c == null) {
    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`)

---

590
6. What the reason is for that status (COLUMN_REASON)

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.

**OK, So Why Is This In This Chapter?**

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:

```java
@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():

```java
@Override
public void onPause() {
    getActivity().unregisterReceiver(onEvent);
    super.onPause();
}
```
The BroadcastReceiver implementation examines the action string of the incoming Intent (via a call to \texttt{getAction()}) and either displays a Toast (for \texttt{ACTION\_NOTIFICATION\_CLICKED}) or enables the start-download Button:

\begin{verbatim}
private BroadcastReceiver onEvent = new BroadcastReceiver()
{
    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);
        }
    }
};
\end{verbatim}

\textbf{What the User Sees}

The user, upon launching the application, sees our three pretty buttons:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{downloaddemo.png}
\caption{The Download Demo Sample, As Initially Launched}
\end{figure}

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):
Sliding down the notification drawer shows the user the progress in the form of a ProgressBar widget:
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:
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:

```
  COLUMN_ID: 12
  COLUMN_BYTES_DOWNLOADED_SO_FAR: 6219229
  COLUMN_LOCAL_URI: file:///mnt/sdcard/Download/test.mp4
  COLUMN_STATUS: 2
  COLUMN_REASON: 0
```
INTENTS, INTENT FILTERS, BROADCASTS, AND BROADCAST RECEIVERS

Tapping the bottom button brings up the activity displaying all downloads, including both successes and failures:

![Downloads - Sorted by date](image)

*Figure 204: The DownloadManager Results*

And, of course, the file is downloaded. In the emulator, our chosen location maps to `/mnt/sdcard/Downloads/test.mp4`.

**Limitations**

DownloadManager works with HTTP URLs, but not HTTPS (SSL) URLs, on Android 2.3. Android 3.0 and newer appear to support HTTPS.

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.

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 BroadcastReceiver 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:

• Only one BroadcastReceiver at a time will receive the broadcast
• The order in which the BroadcastReceiver 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.

We will see ordered broadcasts used in a few places in the book, including for use with one pattern of services letting the user know about work that was accomplished
in the background, where we will see an example implementation of these ordered broadcasts.

**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. We examine this in more detail, along with other “event bus” alternatives, later in the book.
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.

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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book’s GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.

Step #1: Adding a Share Action Bar Item

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.

Modify res/menu/notes.xml to add in the new share toolbar button:

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item>
```

599
Eclipse users can add this via the structured editor for res/menu/notes.xml, following the instructions used for other action bar items.

Note that this menu definition requires a new string resource, named share, with a value like Share.

**Step #2: Sharing the Note**

To actually share the note, we need to start up a new activity using ACTION_SEND. A fragment *could* start up this activity, knowing that the activity is one from a third party and therefore never would be composited within one of our activities by way of fragments. However, to keep things clean, let’s delegate the work for sending the note to the hosting activity, so all startActivity() calls are outside of the fragment.

With that in mind, add the following sendNotes() method to NoteActivity:

```java
def sendNotes(String prose) {
    Intent i = new Intent(Intent.ACTION_SEND);
    i.setType("text/plain");
    i.putExtra(Intent.EXTRA_TEXT, prose);
    startActivity(Intent.createChooser(i, getString(R.string.share_title)));
}
```

We create an ACTION_SEND Intent, fill in our note into EXTRA_TEXT, set the MIME type to be text/plain (since it is unlikely that our user will be entering HTML source code or something as the note), then call startActivity() on the Intent returned by createChooser().
Step #3: Tying Them Together

To tie these pieces together, we need to implement logic to handle our new action bar item and call sendNotes(). To that end, modify the onOptionsItemSelected() implementation on NoteFragment to include the this logic, via an else if block:

```java
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.delete) {
        int position = getArguments().getInt(KEY_POSITION, -1);

        isDeleted = true;
        DatabaseHelper.getInstance(getActivity()).
            deleteNoteAsync(position);

        ((NoteActivity) getActivity()).closeNotes();

        return(true);
    }
    else if (item.getItemId() == R.id.share) {
        ((NoteActivity) getActivity()).sendNotes(editor.getText().toString());

        return(true);
    }

    return(super.onOptionsItemSelected(item));
}
```

All we do is call sendNotes() on the hosting activity, using the current contents of the EditText as the notes, so any not-yet-persisted changes are still shared.

Step #4: Testing the Result

If you run this on a device, navigate to a note, you will see the new action bar item:
If you tap on that, you should get a chooser of various things that know how to send 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.
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 gets garbage-collected, 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 destroys them prematurely. 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
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.

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 `onPause()` and `onStop() 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()`
- `onBind()`, which is called whenever a client binds to the service via `bindService()`
- `onDestroy()`, which is called as the service is being shut down
SERVICES AND THE COMMAND PATTERN

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.

**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 ("WeatherService" or ".WeatherService") to reference our class.

For example, here is a manifest showing the `<service>` element:

```xml
<?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="7"
    android:targetSdkVersion="14"/>
  
  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Sherlock">
    <activity
      android:name="FakePlayer"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
      </intent-filter>
    </activity>
  </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 bi-directional 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 book — 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 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 a while 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.

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:

```java
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);

        return(START_NOT_STICKY);
    }
}
```
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 has to kill off this service (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 player, plus let Android know that the service is delivering part of the foreground experience and therefore should not be shut down.

609
Using the Service

The PlayerFragment demonstrating the use of PlayerService has a very elaborate UI, consisting of two large buttons:

```xml
<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:

```java
package com.commonsware.android.fakeplayer;

import android.content.Intent;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import com.actionbarsherlock.app.SherlockFragment;

public class PlayerFragment extends SherlockFragment 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);

        return(result);
    }

    @Override
}
```
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](https://commonsware.com/computer-sausage/ch19.html), 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.

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.

**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.

**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 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.

Scenario: The Downloader

If you elect to download something from the Play Store, you are welcome to back out of the Market application entirely. This does not cancel the download – the download and installation run to completion, despite no Market 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.

Android 2.3 introduced the DownloadManager (covered in a previous chapter), which would handle this for you. However, you might need that sort of capability on older versions of Android, at least through late 2012, as Android 2.2 fades into the distance.

The sample project reviewed in this section is Service/Downloader.

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 a BroadcastReceiver for this.
Using the Service

The DownloadFragment demonstrating the use of Downloader has a trivial UI, consisting of one large button:

```xml
<?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:

```java
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup parent, Bundle savedInstanceState) {
    View result = inflater.inflate(R.layout.main, parent, false);
    Button b = (Button) result.findViewById(R.id.button);
    b.setOnClickListener(this);

    return result;
}
```

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:

```java
@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.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 {
            try {
                File root = 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();
                }
            }
        }
    }
}
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 `getExternalStorageDirectory()` to find the public folder for downloads. Since this directory may not exist, we need to create it 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

Finally, it would be nice to let somebody know that the download has completed. So, we send a broadcast Intent, with our own custom action (`ACTION_COMPLETE`).

Receiving the Broadcast

Our `DownloadFragment` is set up to listen for that broadcast Intent, by registering a `BroadcastReceiver` in `onResume()` and unregistering it in `onPause()`:
The BroadcastReceiver itself re-enables our button, plus displays a Toast indicating that the download is complete:

```java
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.
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.

This is a rather lengthy 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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book’s GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.
Step #1: Adding a Stub DownloadCheckService

There are a few pieces to our download-the-book-update puzzle:

- 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.

If you wish to make this change using Eclipse's wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

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 following class definition:

```java
define 620
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) {
}

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.

Outside of Eclipse

Create a src/com/commonsware/empublite/DownloadCheckService.java source file, with the content shown above. Also add the following <service> element as a child of the <application> element in your AndroidManifest.xml file:

    <service android:name="DownloadCheckService">
    </service>

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, to the res/menu/options.xml file:

    <item
        android:id="@+id/update"
        android:icon="@android:drawable/ic_menu_save"
        android:showAsAction="ifRoom|withText"
        android:title="@string/download_update">
    </item>

Eclipse users 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 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:
    startActivity(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: Adding a Stub**

**DownloadCompleteReceiver**

Ideally, our actual downloading will be done by DownloadManager, as it handles all of the idiosyncrasies with network type failover and so on. The way we find out that a download from DownloadManager is complete is via a broadcast Intent. So, we need to set up a receiver for that Intent. And, since we do not know if our process will be around when the download is complete, we should set up that BroadcastReceiver in the manifest.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**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 DownloadCompleteReceiver 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 DownloadCompleteReceiver 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 DownloadCompleteReceiver, following the same approach that we used for activities in this application — just be sure to define a receiver instead of an activity.

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 DownloadCompleteReceiver 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, type in android.intent.action.DOWNLOAD_COMPLETE, as this one does not appear in the drop-down in the current version of the ADT plugin for Eclipse.

**Outside of Eclipse**

Create a src/com/commonsware/empublite/DownloadCompleteReceiver.java source file, with the following code:

```java
package com.commonsware.empublite;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class DownloadCompleteReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context ctxt, Intent i) {
    }
}
```

Also add the following `<receiver>` element as a child of the `<application>` element in your AndroidManifest.xml file:

```xml
<receiver android:name="DownloadCompleteReceiver">
    <intent-filter>
        <action android:name="android.intent.action.DOWNLOAD_COMPLETE"/>
    </intent-filter>
</receiver>
```

**Step #4: Completing the DownloadCheckService**

Now that we have some of our other dependencies in place, like DownloadCompleteReceiver, we can add in the business logic for DownloadCheckService.

First, add an UPDATE_URL static data member to DownloadCheckService, containing the URL we will poll to see if there is an update available:
private static final String UPDATE_URL = "http://misc.commonsware.com/empublite-update.json";

Next, replace the stub onHandleIntent() method we have now in DownloadCheckService with the following:

```java
@Override
protected void onHandleIntent(Intent intent) {
    BufferedReader reader = null;
    try {
        URL url = new URL(UPDATE_URL);
        HttpURLConnection c = (HttpURLConnection) url.openConnection();
        c.setRequestMethod("GET");
        c.setReadTimeout(15000);
        c.connect();

        reader = new BufferedReader(new InputStreamReader(c.getInputStream()));

        StringBuilder buf = new StringBuilder();
        String line = null;
        while ((line = reader.readLine()) != null) {
            buf.append(line + "\n");
        }

        checkDownloadInfo(buf.toString());
    }
    catch (Exception e) {
        Log.e(getClass().getSimpleName(), "Exception retrieving update info", e);
    }
    finally {
        if (reader != null) {
            try {
                reader.close();
            }
            catch (IOException e) {
                Log.e(getClass().getSimpleName(), "Exception closing HUC reader", e);
            }
        }
    }
}
```

In this fairly large chunk of code, we are using HttpURLConnection to download UPDATE_URL, streaming the resulting JSON into a StringBuilder. We then pass the String representing the JSON into yet-to-be-implemented checkDownloadInfo()
method. Along the way, we have exception handling to make sure we clean up our socket connection in case something goes wrong.

Then, add an UPDATE_BASEDIR static data member to DownloadCheckService, representing the name of a directory on internal storage where our updates will be stored:

```java
private static final String UPDATE_BASEDIR = "updates";
```

Next, add a getUpdateBaseDir() method to DownloadCheckService that takes UPDATE_BASEDIR and adds it to the getFilesDir() File returned by a Context:

```java
static File getUpdateBaseDir(Context ctxt) {
    return new File(ctxt.getFilesDir(), UPDATE_BASEDIR);
}
```

Then, add an UPDATE_FILENAME static data member to DownloadCheckService, containing the filename to which we will download the update:

```java
public static final String UPDATE_FILENAME = "book.zip";
```

Next, add an PREF_PENDING_UPDATE static data member to DownloadCheckService, containing the key in SharedPreferences where we will store the local location of an in-flight update:

```java
public static final String PREF_PENDING_UPDATE = "pendingUpdateDir";
```

Then, add a pair of string resources:

- update_title, with a value like EmPub Lite Update
- update_description, with a value like A new edition of book content

The JSON in question that we are downloading will be of the form:

```
{"20120512": "http://misc.commonsware.com/WarOfTheWorlds-Update.zip"}
```

With that in mind, add an implementation of checkDownloadInfo() to DownloadCheckService as follows:

```java
private void checkDownloadInfo(String raw) throws JSONException {
    JSONObject json = new JSONObject(raw);
    String version = json.names().getString(0);
    File localCopy = new File(getUpdateBaseDir(this), version);
    if (!localCopy.exists()) {
```
We first parse the JSON and get the version number of the update, which is the value of the one-and-only key of our JSONObject. We then create a File object representing a directory for that update, a subdirectory of our getUpdateBaseDir() directory. If we have already downloaded this update, that directory update's directory will exist by name, and we can skip the download.

Otherwise, we store the directory where we want the update to reside in our SharedPreferences under PREF_PENDING_UPDATE, for later retrieval by another service.

We then configure and enqueue a DownloadManager.Request to have DownloadManager download the update (the value for our version's key in the JSON). The resulting ZIP file is downloaded to external storage, in the standard DIRECTORY_DOWNLOADED location, under the filename represented by UPDATE_FILENAME.

Given this implementation, we need to add three permissions to the manifest:

- android.permissionINTERNET
- android.permission.DOWNLOAD_WITHOUT_NOTIFICATION
Non-Eclipse users can add the following `<uses-permission>` elements as children of the root `<manifest>` element in `AndroidManifest.xml`:

```xml
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.DOWNLOAD_WITHOUT_NOTIFICATION"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

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`. Repeat that process twice more to add the other two permissions listed above.

**Step #5: Adding a Stub DownloadInstallService**

`DownloadManager` will take care of downloading the ZIP file for us. However, once it is downloaded, we need to unZIP it into the desired update directory. And, we cannot do that from a `BroadcastReceiver` triggered by the download being completed, as the unZIP process may take too long.

So, we need another `IntentService` — this one we can call `DownloadInstallService`.

If you wish to make this change using Eclipse's wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**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 `DownloadInstallService` 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 `DownloadInstallService` class.

Then, with `DownloadInstallService` open in the editor, paste in the following class definition:
package com.commonsware.empublite;
import android.app.IntentService;
import android.content.Intent;

public class DownloadInstallService extends IntentService {
    public DownloadInstallService() {
        super("DownloadInstallService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
    }
}

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 DownloadInstallService, following the same approach that we used for DownloadCheckService earlier in this tutorial.

Outside of Eclipse

Create a src/com/commonsware/empublite/DownloadInstallService.java source file, with the content shown above. Also add the following <service> element as a child of the <application> element in your AndroidManifest.xml file:

```xml
<service android:name="DownloadInstallService">
</service>
```

Step #6: Completing the DownloadCompleteReceiver

Our DownloadCompleteReceiver is set up in the manifest to listen for DownloadManager broadcasts. We need to confirm that our update has taken place and, if so, arrange to invoke our DownloadInstallService to unpack it.

With that in mind, replace the stub onReceive() implementation in DownloadCompleteReceiver with the following:

```java
@Override
public void onReceive(Context ctxt, Intent i) {
    File update = new File(
            Environment.getExternalStoragePublicDirectory(
                    Environment.DIRECTORY_DOWNLOADS),
            DownloadCheckService.UPDATE_FILENAME);
```
if (update.exists()) {
    ctxt.startService(new Intent(ctxt, DownloadInstallService.class));
}

We create a File object pointing to where DownloadManager should have downloaded the file, and if the File exists, we send the command to DownloadInstallService.

**Step #7: Completing the DownloadInstallService**

Now, we can unpack our downloaded ZIP file into the desired directory.

First, define three static data members to DownloadInstallService:

- `PREF_UPDATE_DIR`, the key in SharedPreferences where we will store the directory containing a copy of the book that ModelFragment should load from instead of our assets
- `PREF_PREV_UPDATE`, the key in SharedPreferences where we will store the directory containing the previous copy of the book that ModelFragment might presently be using, but can be safely deleted the next time it goes to load up the book contents
- `ACTION_UPDATE_READY`, the name of a broadcast Intent that we will use to alert our running EmPubLiteActivity that an update was completed and that we can now reload the book contents

```java
public static final String PREF_UPDATE_DIR = "updateDir";
public static final String PREF_PREV_UPDATE = "previousUpdateDir";
public static final String ACTION_UPDATE_READY = "com.commonsware.empublite.action.UPDATE_READY";
```

Next, replace our stub `onHandleIntent()` implementation in DownloadInstallService with the following:

```java
@Override
protected void onHandleIntent(Intent intent) {
    SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(this);
    String prevUpdateDir = prefs.getString(PREF_UPDATE_DIR, null);
    String pendingUpdateDir = prefs.getString(DownloadCheckService.PREF_PENDING_UPDATE, null);
    if (pendingUpdateDir != null) {
        File root =
```
Here, we:

- Collect the current update directory (PREF_UPDATE_DIR) and the one that we should be unZIPPING an update into (PREF_PENDING_UPDATE) from SharedPreferences
- Call a to-be-written unzip() method to unZIP the just-downloaded update into the desired destination directory
- Update SharedPreferences to indicate that the just-unZIPPped copy is the update to be used from now on (PREF_UPDATE_DIR) and that the former update directory can be deleted (PREF_PREV_UPDATE)
- Delete the ZIP file, as it is no longer needed
- Send an ACTION_UPDATE_READY ordered broadcast, limited to our package via setPackage(), to let the activity know that our work is done

Finally, add the missing unzip() method to DownloadInstallService:

```java
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) {
```
This is a fairly standard Java unZIP-the-whole-ZIP-file implementation, though it does use the Android-recommended `sync()` approach to ensure that our disk writes are flushed.

**Step #8: Updating ModelFragment**

ModelFragment needs to know to load our downloaded update, instead of assets, when that update is available. To that end, modify `doInBackground()` of the `ContentsLoadTask` inner class of `ModelFragment` to look like this:

```java
@Override
protected Void doInBackground(Context... ctxt) {
    String updateDir = prefs.getString(DownloadInstallService.PREF_UPDATE_DIR, null);

    try {
        StringBuilder buf = new StringBuilder();
        InputStream json = null;

        if (updateDir != null && new File(updateDir).exists()) {
            json = new FileInputStream(new File(new File(updateDir), "contents.json"));
        } else {
            json = ctxt[0].getAssets().open("book/contents.json");
        }
```
BufferedReader in =
        new BufferedReader(new InputStreamReader(json));
String str;
while (str = in.readLine() != null) {
    buf.append(str);
}
in.close();
if (updateDir != null && new File(updateDir).exists()) {
    localContents =
            new BookContents(new JSONObject(buf.toString()),
            new File(updateDir));
} else {
    localContents =
            new BookContents(new JSONObject(buf.toString()));
}
catch (Exception e) {
    this.e = e;
}

String prevUpdateDir =
prefs.getString(DownloadInstallService.PREF_PREV_UPDATE, null);
if (prevUpdateDir != null) {
    File toBeDeleted = new File(prevUpdateDir);
    if (toBeDeleted.exists()) {
        deleteDir(toBeDeleted);
    }
}
return(null);

The differences are:

- We read the PREF_UPDATE_DIR preference out of prefs
- If the update directory is not null and that directory actually exists, we load
  the JSON out of it instead of out of assets
- If the update directory is not null and that directory actually exists, we tell
  the BookContents to use that directory instead of assets
- We see if there is a value for PREF_PREV_UPDATE, and if the value and the
  pointed-to directory exists, we delete that directory using a to-be-
  implemented deleteDir() method

632
This requires revisions to the data members, constructor, and `getChapterFile()` method of `BookContents`, to support a new `updateDir` value:

```java
package com.commonsware.empublite;

import android.net.Uri;
import java.io.File;
import org.json.JSONArray;
import org.json.JSONObject;

public class BookContents {
    JSONObject raw=null;
    JSONArray chapters;
    File updateDir=null;

    BookContents(JSONObject raw) {
        this(raw, null);
    }

    BookContents(JSONObject raw, File updateDir) {
        this.raw=raw;
        this.updateDir=updateDir;
        chapters=raw.optJSONArray("chapters");
    }

    int getChapterCount() {
        return(chapters.length());
    }

    String getChapterFile(int position) {
        JSONObject chapter=chapters.optJSONObject(position);

        if (updateDir != null) {
            return(Uri.fromFile(new File(updateDir,
                chapter.optString("file"))).toString());
        }

        return("file:///android_asset/book/"+chapter.optString("file"));
    }

    String getTitle() {
        return(raw.optString("title"));
    }
}
```

This also requires that we add the `deleteDir()` method to `ModelFragment`:

```java
private static boolean deleteDir(File dir) {
    if (dir.exists() && dir.isDirectory()) {
        File[] children=dir.listFiles();

        for (File child : children) {
            ...
        }

        dir.delete();
    }
    return(false);
}
```
boolean ok = deleteDir(child);

if (!ok) {
    return(false);
}

return(dir.delete());

Also, we now have a dependency: ContentsLoadTask needs the preferences that are loaded by PrefsLoadTask. Hence, we can no longer launch these in parallel, but instead must wait on executing the ContentsLoadTask until after PrefsLoadTask is done. This is a surprisingly simple change to deliverModel() in ModelFragment, converting the if (contents == null && contentsTask == null) check to be an else if, chaining to the previous if:

```java
synchronized private void deliverModel() {
    if (prefs != null && contents != null) {
        ((EmPubLiteActivity) getActivity()).setupPager(prefs, contents);
    } else {
        if (prefs == null && prefsTask == null) {
            prefsTask = new PrefsLoadTask();
            executeAsyncTask(prefsTask,
                getActivity().getApplicationContext());
        } else if (contents == null && contentsTask == null) {
            contentsTask = new ContentsLoadTask();
            executeAsyncTask(contentsTask,
                getActivity().getApplicationContext());
        } else {
            // Do something with old prefsTask and contentsTask...
        }
    }
}
```

**Step #9: Adding a BroadcastReceiver to EmPubLiteActivity**

We also need to catch that broadcast from DownloadInstallService and arrange to reload our book contents once the update is complete.

To do this, in ModelFragment, move the contents of the else if block in deliverModel() to a separate method, named updateBook():

```java
void updateBook() {
    contentsTask = new ContentsLoadTask();
}
```
executeAsyncTask(contentsTask, getActivity().getApplicationContext());

Then, have deliverModel() use updateBook():

synchronized private void deliverModel() {
    if (prefs != null && contents != null) {
        ((EmPubLiteActivity) getActivity()).setupPager(prefs, contents);
    } else {
        if (prefs == null && prefsTask == null) {
            prefsTask = new PrefsLoadTask();
            executeAsyncTask(prefsTask, getActivity().getApplicationContext());
        } else if (contents == null && contentsTask == null) {
            updateBook();
        }
    }
}

In EmPubLiteActivity, add a BroadcastReceiver data member named onUpdate that will call updateBook() on the ModelFragment, then abort the ordered broadcast:

private BroadcastReceiver onUpdate = new BroadcastReceiver() {
    public void onReceive(Context ctxt, Intent i) {
        model.updateBook();
        abortBroadcast();
    }
};

Then, register that receiver in onResume() of EmPubLiteActivity, by adding these lines at the end of onResume():

@Override
public void onResume() {
    super.onResume();

    if (prefs != null) {
        pager.setKeepScreenOn(prefs.getBoolean(PREF_KEEP_SCREEN_ON, false));
    }

    IntentFilter f = new IntentFilter(DownloadInstallService.ACTION_UPDATE_READY);
    f.setPriority(1000);
    registerReceiver(onUpdate, f);
}

We are setting the priority to be 1000 in preparation for an upcoming tutorial.
Finally, unregister that receiver by adding the following line to the top of `onPause()`:

```java
unregisterReceiver(onUpdate);
```

We have one lingering problem: our `BroadcastReceiver` is referring to a model data member that does not exist. That is our `ModelFragment`. Heretofore, we have not needed to call `ModelFragment` from `EmPubLiteActivity`, but now we do, in order to have `ModelFragment` reload the book.

So, add a model data member to `EmPubLiteActivity`:

```java
private ModelFragment model=null;
```

Then, adjust the `onCreate()` implementation in `EmPubLiteActivity` to assign a value to `model`, whether we create a new `ModelFragment` or access the one we created earlier when the activity was first created:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (getSupportFragmentManager().findFragmentByTag(MODEL) == null) {
        model=new ModelFragment();
        getSupportFragmentManager().beginTransaction().add(model, MODEL)
            .commit();
    } else {
        model=
            (ModelFragment)getSupportFragmentManager().findFragmentByTag(MODEL);
    }

    setContentView(R.layout.main);

    pager=(ViewPager)findViewByDetails(R.id.pager);
    getSupportActionBar().setHomeButtonEnabled(true);
}
```

We also have one other tweak to make. `ContentsAdapter` used to have the responsibility of adding the file: `android_asset/book/` to the path returned by `BookContents`. That is no longer valid, as `BookContents` returns the full path (whether local or to an asset). So, change `getItem()` in `ContentsAdapter` to be:

```java
@Override
public Fragment getItem(int position) {
    return(SimpleContentFragment.newInstance(contents.getChapterFile(position)));
}
```
At this point, if you build and run the app, you will see the update action bar item (looks like a floppy disk):

![The New Action Bar Item](image)

Pressing that and waiting a moment should cause your book to be updated with new contents downloaded from the Internet:
Step #10: Discussing the Flaws

The tutorials in this book are not meant to be production-grade code. That being said, the approaches we are taking in this specific tutorial are weaker than usual.

Notably, the way we have set up DownloadCompleteReceiver will cause it to receive broadcasts for any use of DownloadManager. There is no good way to have DownloadManager only tell us about our downloads. However, we could use some more advanced techniques to have DownloadCompleteReceiver be disabled except during the window of time when we are performing the actual download.

We also do not take any steps to limit the downloads. If the user taps the action bar item twice, we might happily kick off two downloads.

In Our Next Episode…

... we will update the book’s contents ourselves, periodically in the background.
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 is 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:

```java
public void onCreate(Bundle icicle) {
    // other work here

    someWidget.postDelayed(everySecond, 1000);
}

Runnable everySecond = new Runnable() {
    public void run() {
        // do periodic work
        anyOldWidget.postDelayed(everySecond, 1000);
    }
};
```
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 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.
ALARM MANAGER AND THE SCHEDULED SERVICE PATTERN

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.

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.

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:

```java
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) {
        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);
    }

    @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
**ALARM MANAGER AND THE SCHEDULED SERVICE PATTERN**

- 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

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 Pending Intent 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 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

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).

### 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 will examine this process a bit later in this chapter.

### 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 `SharedPreferences`. 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:

```java
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).

## Get Moving, First Thing

If you want to establish your alarms at boot time, to cope with a reboot wiping out your alarm schedule, you will need to arrange to have a BroadcastReceiver get control at boot time.

### The Permission

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.

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.
The Receiver Element

There are two ways you can receive a broadcast Intent. One is to use registerReceiver() from an existing Activity, Service, or ContentProvider. The other is to register your interest in the Intent in the manifest in the form of a <receiver> element:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest
    android:versionCode="1"
    android:versionName="1.0"
    package="com.commonsware.android.sysevents.boot"
    xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-sdk
        android:minSdkVersion="3"
        android:targetSdkVersion="6" />
    <supports-screens
        android:largeScreens="false"
        android:normalScreens="true"
        android:smallScreens="false" />
    <uses-permission
        android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
    <application
        android:icon="@drawable/cw"
        android:label="@string/app_name">
        <receiver
            android:name=".OnBootReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

The above AndroidManifest.xml, from the SystemEvents/OnBoot sample project, shows that we have registered a broadcast receiver named OnBootReceiver, set to be given control when the android.intent.action.BOOT_COMPLETED Intent is broadcast.

In this case, we have no choice but to implement our receiver this way — by the time any of our other components (e.g., an Activity) were to get control and be able to call registerReceiver(), the BOOT_COMPLETED Intent will be long gone.

The Receiver Implementation

Now that we have told Android that we would like to be notified when the boot has completed, and given that we have been granted permission to do so by the user, we now need to actually do something to receive the Intent. This is a simple matter of creating a BroadcastReceiver, such as seen in the OnBootReceiver implementation shown below:
A BroadcastReceiver is not a Context, and so it gets passed a suitable Context object in `onReceive()` to use for accessing resources and the like. The `onReceive()` method also is passed the Intent that caused our BroadcastReceiver to be created, in case there are “extras” we need to pull out (none in this case).

In `onReceive()`, we can do whatever we want, subject to some limitations:

- We are not a Context, like an Activity, so we cannot directly modify the UI
- If we want to do anything significant, it is better to delegate that logic to a service that we start from here (e.g., calling `startService()` on the supplied Context) rather than actually doing it here, since BroadcastReceiver implementations need to be fast
- We cannot start any background threads, directly or indirectly, since the BroadcastReceiver gets discarded as soon as `onReceive()` returns

In this case, we simply log the fact that we got control.

To test this, install it on an emulator (or device), shut down the emulator, then restart it.

**New Behavior With Android 3.1**

It used to be that Android applications registering a `BOOT_COMPLETED` BroadcastReceiver would get control at boot time. Starting with Android 3.1, that may or may not occur.

If you install an application that registers a `BOOT_COMPLETED` receiver, and simply restart the Android 3.1 device, the receiver does not get control at boot time. It appears that the user has to start up an activity in that application first (e.g., from
the launcher) before Android will deliver a BOOT_COMPLETED Intent to that application.

Google has long said that users should launch an activity from the launcher first, before that application can do much. Preventing BOOT_COMPLETED from being delivered until the first activity is launched is a logical extension of the same argument.

Most apps will be OK with this change. For example, if your boot receiver is there to establish an AlarmManager schedule, you also needed to establish that schedule when the app is first run, so the user does not have to reboot their phone just to set up your alarms. That pattern does not change – it is just that if the user happens to reboot the phone, it will not set up your alarms, until the user runs one of your activities.

**Archetype: Scheduled Service Polling**

Given that we now know how to get control at boot time, we can return our attention to AlarmManager.

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:
**ALARM MANAGER AND THE SCHEDULED SERVICE PATTERN**

- 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](https://commonsware.com).

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

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
<?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-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />

  <uses-sdk android:minSdkVersion="7"
            android:targetSdkVersion="11" />

  <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" />
      </intent-filter>
    </activity>
  </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:

```java
package com.commonsware.android.schedsvc;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
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:

```java
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.

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:

```java
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 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
dfork 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. You can download it as an Android library project or as a JAR from a GitHub
repository. It is open source, licensed under the Apache License 2.0.

WakefulIntentService allows you to implement “the handoff pattern”:

• You add the JAR 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

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.

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 we can make use of WakefulIntentService in our code.

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:

```java
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.

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():

```java
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 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-wake up 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:

```
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
    operation=PendingIntent{42e15d20: PendingIntentRecord{42e0cc28
      com.google.android.gms startService}}
  RTC_WAKEUP #9: Alarm{42787d10 type 0 com.rememberthemilk.MobileRTM}
    type=0 when=+16h10m13s480ms repeatInterval=0 count=0
    operation=PendingIntent{426068a0: PendingIntentRecord{42787c70
      com.rememberthemilk.MobileRTM broadcastIntent}}
```
ALARM MANAGER AND THE SCHEDULED SERVICE PATTERN

RTC_WAKEUP #8: Alarm{43422428 type 0 com.google.android.calendar}
type=0 when=+11h49m38s572ms repeatInterval=0 count=0
operation=PendingIntent{425edd68: PendingIntentRecord{42d6f460
com.google.android.calendar broadcastIntent}}
RTC_WAKEUP #7: Alarm{425eca98 type 0 com.google.android.gms}
type=0 when=+11h14m46s303ms repeatInterval=0 count=0
operation=PendingIntent{425e2cf0: PendingIntentRecord{42d2f0d8
com.google.android.gms broadcastIntent}}
RTC_WAKEUP #6: Alarm{434c2458 type 0 android}
type=0 when=+10h41m56s911ms repeatInterval=0 count=0
operation=PendingIntent{42772408: PendingIntentRecord{42e201e8 android
broadcastIntent}}
RTC_WAKEUP #5: Alarm{42a45908 type 0 com.google.android.partnersetup}
type=0 when=+9h8m56s565ms repeatInterval=0 count=0
operation=PendingIntent{425a9df0: PendingIntentRecord{42d586f8
com.google.android.partnersetup startService}}
RTC_WAKEUP #4: Alarm{42cee778 type 0 com.android.vending}
type=0 when=+8h14m34s283ms repeatInterval=0 count=0
operation=PendingIntent{42d49f20: PendingIntentRecord{42e894e8
com.android.vending startService}}
RTC_WAKEUP #3: Alarm{43436e28 type 0 com.google.android.gsf}
type=0 when=+3h41m38s284ms repeatInterval=40528000 count=0
operation=PendingIntent{42620810: PendingIntentRecord{42d618f8
com.google.android.gsf broadcastIntent}}
RTC_WAKEUP #2: Alarm{42c55cc8 type 0 com.google.android.apps.genie.geniewidget}
type=0 when=+9h4m56s911ms repeatInterval=40528000 count=0
operation=PendingIntent{42620810: PendingIntentRecord{42d618f8
com.google.android.apps.genie.geniewidget broadcastIntent}}
RTC_WAKEUP #1: Alarm{42d6b2a8 type 0 com.google.android.gsf}
type=0 when=+11m57s390ms repeatInterval=1800000 count=0
operation=PendingIntent{42d546e0: PendingIntentRecord{42d546e0
com.google.android.gsf broadcastIntent}}
RTC #4: Alarm{433ee498 type 1 com.google.android.calendar}
type=1 when=+16h10m8s480ms repeatInterval=0 count=0
operation=PendingIntent{42604c48: PendingIntentRecord{42d963b0
com.google.android.calendar broadcastIntent}}
RTC #3: Alarm{42e69130 type 1 android}
type=1 when=+4h10m8s480ms repeatInterval=0 count=0
operation=PendingIntent{428b7d08: PendingIntentRecord{428f77d0 android
broadcastIntent}}
RTC #2: Alarm{4345ddd0 type 1 com.evernote}
type=1 when=+42m11s207ms repeatInterval=3600000 count=1
operation=PendingIntent{42a865a0: PendingIntentRecord{4303ce70 com.evernote
startService}}
RTC #1: Alarm{42d61370 type 1 com.rememberthemilk.MobileRTM}
type=1 when=+15m27s587ms repeatInterval=0 count=0
operation=PendingIntent{42d4ad88: PendingIntentRecord{42d611c8
com.rememberthemilk.MobileRTM broadcastIntent}}
RTC #0: Alarm{433dfdd8 type 1 com.google.android.deskclock}
type=1 when=+10m9s706ms repeatInterval=900000 count=0
operation=PendingIntent{42d9e198: PendingIntentRecord{42ded468
com.google.android.deskclock broadcastIntent}}

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{42de220c: PendingIntentRecord{42ace73e8
com.google.android.apps.maps broadcastIntent}}
ELAPSED_WAKEUP #15: Alarm{42ca4638 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}}
ELAPSED_WAKEUP #14: Alarm{42ccf23f0 type 2 com.google.android.location
  type=2 when=+2h38m35s113ms repeatInterval=0 count=0
  operation=PendingIntent{42a33658: PendingIntentRecord{42d5c5a8
com.google.android.location broadcastIntent}}
ELAPSED_WAKEUP #13: Alarm{43ac42000 type 2 com.google.android.location
  type=2 when=+13h34m38s568ms repeatInterval=0 count=0
  operation=PendingIntent{4302f9b60: PendingIntentRecord{42d6fca8
com.google.android.location broadcastIntent}}
ELAPSED_WAKEUP #12: Alarm{42f8c8d0 type 2 android
  type=2 when=+3h10m8s458ms repeatInterval=0 count=0
  operation=PendingIntent{42e1b670: PendingIntentRecord{42e2c2d0
com.google.android.apps.maps broadcastIntent}}
ELAPSED_WAKEUP #11: Alarm{4282a7d0 type 2 com.google.android.apps.maps
  type=2 when=+3h10m8s458ms repeatInterval=0 count=0
  operation=PendingIntent{42e1b670: PendingIntentRecord{42e2c2d0
com.google.android.apps.maps broadcastIntent}}
ELAPSED_WAKEUP #10: Alarm{42dc89bf0 type 2 com.tripit
  type=2 when=+2h49m31s664ms repeatInterval=0 count=0
  operation=PendingIntent{4280e408: PendingIntentRecord{42db5dc8
android broadcastIntent}}
ELAPSED_WAKEUP #9: Alarm{42e57820 type 2 com.google.android.apps.maps
  type=2 when=+2h45m10s324ms repeatInterval=0 count=0
  operation=PendingIntent{42a364c0: PendingIntentRecord{42dbf9a0
com.google.android.apps.maps broadcastIntent}}
ELAPSED_WAKEUP #8: Alarm{435057c0 type 2 com.google.android.apps.maps
  type=2 when=+2h16m5s664ms repeatInterval=0 count=0
  operation=PendingIntent{42a364c0: PendingIntentRecord{42db5dc8
com.google.android.apps.maps broadcastIntent}}
ELAPSED_WAKEUP #7: Alarm{43065540 type 2 com.google.android.apps.maps
  type=2 when=+1h17m5s685ms repeatInterval=0 count=0
  operation=PendingIntent{42e348f18: PendingIntentRecord{4289b130
android broadcastIntent}}
ELAPSED_WAKEUP #6: Alarm{43348f18 type 2 android
  type=2 when=+54m8s461ms repeatInterval=0 count=0
  operation=PendingIntent{428e7520: PendingIntentRecord{4289b130
android broadcastIntent}}
ELAPSED_WAKEUP #5: Alarm{42990000 type 2 android
  type=2 when=+24m57s328ms repeatInterval=3600000 count=1
  operation=PendingIntent{42d3140: PendingIntentRecord{42ad0140
android broadcastIntent}}
ELAPSED_WAKEUP #4: Alarm{430000e8 type 2 com.google.android.apps.maps
  type=2 when=+9m57s328ms repeatInterval=900000 count=1

661
**ALARM MANAGER AND THE SCHEDULED SERVICE PATTERN**

```
operation=PendingIntent{42a69660: PendingIntentRecord{42d2d518
com.google.android.apps.maps startService}}
ELAPSED_WAKEUP #3: Alarm{42c4a6e0 type 2 com.google.android.apps.maps}
  type=2 when=+5m52s371ms repeatInterval=0 count=0
  operation=PendingIntent{42abf1f0: PendingIntentRecord{42992450
com.google.android.apps.maps broadcastIntent}}
ELAPSED_WAKEUP #2: Alarm{42d60808 type 2 com.google.android.gsf}
  type=2 when=+4m10s512ms repeatInterval=0 count=0
  operation=PendingIntent{430495a8: PendingIntentRecord{4342bc08
com.google.android.gsf broadcastIntent}}
ELAPSED_WAKEUP #1: Alarm{42b8a490 type 2 com.android.phone}
  type=2 when=+1m38s451ms repeatInterval=0 count=0
  operation=PendingIntent{42e8bde0: PendingIntentRecord{42a0d700: PendingIntentRecord{43496428
com.android.phone broadcastIntent}}
ELAPSED_WAKEUP #0: Alarm{42d4cd98 type 3 android}
  type=3 when=+9m4s957ms repeatInterval=0 count=0
  operation=PendingIntent{427a6ab8: PendingIntentRecord{428f4558 android
broadcastIntent}}
```

Broadcast ref count: 0

Top Alarms:
+14m24s97ms running, 0 wakeups, 9567 alarms: android
  act=android.intent.action.TIME_TICK
+1m5s72ms running, 4890 wakeups, 4890 alarms: com.android.phone
  act=com.android.server.sip.SipWakeupTimer@42626830
+1m3s465ms 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

662
### AlarmManager and the Scheduled Service Pattern

<table>
<thead>
<tr>
<th>Duration</th>
<th>Wakeups</th>
<th>Alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>+42s830ms</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>+35s479ms</td>
<td>0</td>
<td>954</td>
</tr>
<tr>
<td>+14s28ms</td>
<td>1609</td>
<td>1609</td>
</tr>
<tr>
<td>+11s98ms</td>
<td>171</td>
<td>171</td>
</tr>
<tr>
<td>+8s380ms</td>
<td>893</td>
<td>893</td>
</tr>
<tr>
<td>+8s353ms</td>
<td>569</td>
<td>569</td>
</tr>
<tr>
<td>+7s734ms</td>
<td>0</td>
<td>957</td>
</tr>
<tr>
<td>+2s211ms</td>
<td>0</td>
<td>320</td>
</tr>
<tr>
<td>+1s144ms</td>
<td>151</td>
<td>151</td>
</tr>
<tr>
<td>+9ms</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+3ms</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>+922ms</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>+479ms</td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>+10ms</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>+100ms</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>+9ms</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+3ms</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>+928ms</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>+6ms</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+934ms</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>+6ms</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+14s742ms</td>
<td>911</td>
<td></td>
</tr>
<tr>
<td>+8s353ms</td>
<td>569</td>
<td></td>
</tr>
<tr>
<td>+2s211ms</td>
<td>85</td>
<td></td>
</tr>
</tbody>
</table>

Alarm Stats:
- com.google.android.location +120ms running, 12 wakeups:
- +73ms 7 wakes 7 alarms:
- +47ms 5 wakes 5 alarms:

Alarm Manager and the Scheduled Service Pattern

- act=com.android.server.ThrottleManager.action.POLL
- act=com.android.internal.telephony.gprs-data-stall
- act=com.android.providers.calendar.intent.CalendarProvider2
- act=android.content.syncmanager.SYNC_ALARM
- act=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:
    - +14m24s897ms 0 wakes 9567 alarms: act=android.intent.action.TIME_TICK
    - +1m13s465ms 0 wakes 320 alarms:

- 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.tripit +934ms running, 51 wakes:
  - +928ms 50 wakes 50 alarms: act=com.tripit.PARTIAL_TRIP_REFRESH

  - +6ms 1 wakes 1 alarms: act=com.tripit.FULL_TRIP_REFRESH

- com.google.android.apps.maps
  - +14s742ms running, 911 wakes:
  - +8s353ms 569 wakes 569 alarms: cmp={com.google.android.apps.maps/com.google.googlenav.prefetch.android.PrefetcherService}

- act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_LOCATOR
### ALARM MANAGER AND THE SCHEDULED SERVICE PATTERN

<table>
<thead>
<tr>
<th>Activity</th>
<th>Duration</th>
<th>Wakes</th>
<th>Alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_SENSOR_UPLOADER</code></td>
<td>+1s206ms</td>
<td>103</td>
<td>103</td>
</tr>
<tr>
<td><code>act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_BURST_COLLECTION_TRIGGER</code></td>
<td>+807ms</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><code>act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_S_COLLECTOR</code></td>
<td>+759ms</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td><code>act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_CACHE_UPDATER</code></td>
<td>+566ms</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td><code>act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_IN_OUT_DOOR_COLLECTOR</code></td>
<td>+385ms</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td><code>act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_ACTIVE_COLLECTOR</code></td>
<td>+308ms</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td><code>act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_PASSIVE_COLLECTOR</code></td>
<td>+77ms</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td><code>act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_ACTIVITY_DETECTION</code></td>
<td>+42ms</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><code>act=com.google.android.apps.maps.nlp.ALARM_WAKEUP_CALIBRATION_COLLECTOR</code></td>
<td>+28ms</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@42626830</code></td>
<td>+1m15s</td>
<td>4890</td>
<td>4890</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@42858508</code></td>
<td>+42s</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+727ms</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+727ms</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+724ms</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+724ms</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+705ms</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+705ms</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+630ms</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+630ms</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+616ms</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+603ms</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+603ms</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+576ms</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td><code>act=com.android.server.sip.SipWakeupTimer@428657c8</code></td>
<td>+576ms</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td><code>act=com.google.android.apps.genie.geniewidget</code></td>
<td>+397ms</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>act=com.google.android.calendar.APPWIDGET_SCHEDULED_UPDATE</code></td>
<td>+332ms</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td><code>act=com.google.android.calendar.APPWIDGET_SCHEDULED_UPDATE</code></td>
<td>+65ms</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Subscribe to updates at https://commonsware.com

Special Creative Commons BY-NC-SA 4.0 License Edition
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.
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

### 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 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()`:

```java
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;

public class PollReceiver extends WakefulBroadcastReceiver {
    private static final int PERIOD=900000; // 15 minutes
    private static final int INITIAL_DELAY=5000; // 5 seconds
    ```
ALARM MANAGER AND THE SCHEDULED SERVICE PATTERN

@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() {
        super("ScheduledService");
    }
    @Override

667
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 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.
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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book’s GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.

**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.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).
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 permission.

Outside of Eclipse

Create a src/com/commonsware/empublite/UpdateReceiver.java source file, with the following Java code:

```java
package com.commonsware.empublite;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class UpdateReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context ctxt, Intent i) {
    }
}
```

TUTORIAL #17 - PERIODIC BOOK UPDATES
Then, add the following `<receiver>` element as a child of the `<application>` element in `AndroidManifest.xml`:

```xml
<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:

```xml
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
```

**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`:

```java
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`):

```java
@Override
public void onReceive(Context ctxt, Intent i) {
    if (i.getAction() != null) {
        scheduleAlarm(ctxt);
    }
}
```

Finally, at the end of `onCreate()` of `EmPubLiteActivity`, add:

```java
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, the user will not be using their device. Therefore, it is possible that the device will fall asleep while we try to download the update. Therefore, we need to switch to using `WakefulIntentService`.

Visit the [WakefulIntentService releases](https://commonsware.com) and download the `CWAC-WakefulIntentService.jar` file listed there. Put it in the `libs/` directory of your project, creating that directory if it does not exist. Eclipse users 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

Then, modify `DownloadCheckService` and `DownloadInstallService` to inherit from `com.commonsware.cwac.wakeful.WakefulIntentService` instead of from `IntentService`. This will cause you to need to rename your `onHandleIntent()` methods to be `doWakefulWork()`.
Also, add the \texttt{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 \texttt{<uses-permission>} element.

**Step \#4: Using WakefulIntentService**

To correctly use \texttt{WakefulIntentService}, we need to use \texttt{sendWakefulWork()} to send commands to one, rather than \texttt{startService()}.

With that in mind, in \texttt{EmPubLiteActivity}, change the \texttt{R.id.update} case of the \texttt{switch} statement in \texttt{onOptionsItemSelected()} to use \texttt{sendWakefulWork()}:

```java
    case R.id.update:
        WakefulIntentService.sendWakefulWork(this, DownloadCheckService.class);
        return (true);
```

Similarly, in \texttt{DownloadCompleteReceiver}, change \texttt{onReceive()} to use \texttt{sendWakefulWork()}:

```java
package com.commonsware.empublite;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Environment;
import java.io.File;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class DownloadCompleteReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context ctxt, Intent i) {
        File update =
            new File(Environment.getExternalStoragePublicDirectory(
                        Environment.DIRECTORY_DOWNLOADS),
                        DownloadCheckService.UPDATE_FILENAME);

        if (update.exists()) {
            WakefulIntentService.sendWakefulWork(ctxt, DownloadInstallService.class);
        }
    }
}
```
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:

```java
@Override
public void onReceive(Context ctxt, Intent i) {
    if (i.getAction() != null) {
        scheduleAlarm(ctxt);
    } else {
        WakefulIntentService.sendWakefulWork(ctxt,
                                              DownloadCheckService.class);
    }
}
```

To test this:

1. In your device or emulator, uninstall the existing EmPubLite application, (e.g., by using the Settings app)
2. Install and run the revised app in your device or emulator, and confirm that you are viewing the non-updated book, then press BACK to exit the activity
3. Temporarily modify the time of your device 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. Find something to pass the time for those few minutes, such as procuring liquid refreshment suitable for the time of day and locale
5. A few minutes after 4am, run the app and confirm that you have downloaded the updated app, then fix your device or emulator’s clock back to normal

In Our Next Episode…

... we will let the user know about updates from the background via a Notification.
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 a phone, they will appear in the status bar, on the top of the screen, left-aligned:
NOTIFICATIONS

On a tablet, they will appear in the system bar, on the bottom of the screen, towards the lower-right corner:

In either case, you can expand the “notification drawer” to get more details about the active notifications, either by sliding down the status bar:
or by tapping on the clock on the system bar:

![Notification Drawer, on a Galaxy Tab 2](image)

*Figure 211: Notification Drawer, on a Galaxy Tab 2*

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

```java
public class DownloadFragment extends SherlockFragment implements View.OnClickListener {
  private Button b=null;

  @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);
  }

  @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:

```java
b.setAutoCancel(true).setDefaults(Notification.DEFAULT_ALL)
    .setWhen(System.currentTimeMillis());

if (e == null) {
    b.setTitle(getString(R.string.download_complete))
        .setMessage(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.setTitle(getString(R.string.exception))
        .setMessage(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. If you are supporting older devices, the Android Support package has a NotificationCompat.Builder backport of the same functionality, and that is what we are using in this particular project.

We can call methods on the Builder to configure the Notification that we want to display. Whether our download succeeded or failed, we use three 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
NOTIFICATIONS

- `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.
- `setWhen(System.currentTimeMillis())` associates the current time with the Notification, which may be displayed in the notification drawer for this notification (depending on device configuration).

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.
- `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:

```xml
<uses-permission android:name="android.permission.VIBRATE"/>
```

Running this in a device or emulator will display the Notification upon completion of the download:

![Sample Notification on a Galaxy Nexus](image)

*Figure 212: Sample Notification, on a Galaxy Nexus*

Opening the notification drawer displays our Notification details:
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.

### Seeking Some Order

By default, broadcasts are sent more or less in parallel. If there are ten BroadcastReceiver objects that will all qualify for an Intent via their IntentFilter, all ten will get the broadcast, in an indeterminate order, some possibly at the same time.

Sometimes, this is not what we want. We want broadcasts to be picked up serially, in a known sequence of possible receivers. That can be handled in Android via an ordered broadcast. This is particularly important for situations where we are using AlarmManager in the background, so we can update either the foreground activity or raise a Notification if we do not have an activity in the foreground.
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 ordered broadcast.

The recipe for the Activity-or-Notification pattern is:

1. Define an action string you will use when the event occurs that you want to go to the activity or notification (e.g., com.commonsware.java.packages.are.fun.EVENT).
2. Dynamically register a BroadcastReceiver in your activity, with an IntentFilter set up for the aforementioned action string and with a positive priority (the default priority for a filter is 0). This receiver should then have the activity do whatever it needs to do to update the UI based on this event. The receiver should also call abortBroadcast() to prevent others from getting it. Be sure to register the receiver in onStart() or onResume() and unregister the receiver in the corresponding onStop() or onPause() method.
3. Register in your manifest a BroadcastReceiver, with an <intent-filter> set up for the aforementioned action string. This receiver should raise the Notification.
4. In your service (e.g., an IntentService), when the event occurs, call sendOrderedBroadcast().

And that’s it. Android takes care of the balance. If the activity is on-screen, its receiver will be registered, so it will get the event, process it, and cancel the
broadcast. If the activity is not on-screen, its receiver will not be registered, so the event will go to the default handler, in the form of your manifest-registered BroadcastReceiver, which will raise the Notification.

For example, let's take a look at the Notifications/Ordered sample application.

In our OrderedActivity, in onCreate(), we set up AlarmManager to pass control to a service (NoticeService) every five seconds:

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    notice=(Button) findViewById(R.id.notice);

    ((NotificationManager) getSystemService(NOTIFICATION_SERVICE))
       .cancelAll();

    mgr=(AlarmManager) getSystemService(Context.ALARM_SERVICE);

    Intent i=new Intent(this, NoticeService.class);

    pi=PendingIntent.getService(this, 0, i, 0);

    cancelAlarm(null);

    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                     SystemClock.elapsedRealtime()+1000,
                     5000,
                     pi);
}
```

We also rig up a button to cancel that alarm when pressed, via a cancelAlarm() method:

```java
public void cancelAlarm(View v) {
    mgr.cancel(pi);
}
```

The NoticeService, when invoked by the AlarmManager, should theoretically do some work. In our case, doing work sounds too much like doing work, and we are lazy in this sample, so we skip straight to sending the ordered broadcast:

```java
package com.commonsware.android.ordered;

import android.app.IntentService;
import android.content.Intent;
```
public class NoticeService extends IntentService {
    public static final String BROADCAST =
            "com.commonsware.android.ordered.NoticeService.BROADCAST";
    private static Intent broadcast = new Intent(BROADCAST);

    public NoticeService() {
        super("NoticeService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        sendOrderedBroadcast(broadcast, null);
    }
}

OrderedActivity, in onResume(), registers a BroadcastReceiver to handle this broadcast, with a high-priority IntentFilter:

    @Override
    public void onResume() {
        super.onResume();

        IntentFilter filter = new IntentFilter(NoticeService.BROADCAST);
        filter.setPriority(2);
        registerReceiver(onNotice, filter);
    }

We unregister that receiver in onPause():

    @Override
    public void onPause() {
        super.onPause();

        unregisterReceiver(onNotice);
    }

The BroadcastReceiver itself updates the caption of our Button with the current date and time:

    private BroadcastReceiver onNotice = new BroadcastReceiver() {
        public void onReceive(Context ctxt, Intent i) {
            notice.setText(new Date().toString());
            abortBroadcast();
        }
    };

The BroadcastReceiver also aborts the broadcast, so no other receivers could get it.
Hence, if we start up the activity and let it run, our Button caption simply changes every five seconds:

![Figure 214: The OrderedActivity, showing the time of the last alarm](image)

But, what happens if we leave the activity, such as via BACK or HOME?

In that case, we also have a `<receiver>` element in our manifest, set up to listen for the same broadcast:

```xml
<?xml version="1.0" encoding="utf-8"?
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.commonsware.android.ordered"
   android:versionCode="1"
   android:versionName="1.0">
   <supports-screens
      android:largeScreens="true"
      android:normalScreens="true"
      android:smallScreens="false"/>
   <uses-sdk
      android:minSdkVersion="7"
      android:targetSdkVersion="11"/>
   <uses-permission android:name="android.permission.VIBRATE"/>
</manifest>
```
That is tied to a `NoticeReceiver` that simply displays a Notification:

```java
package com.commonsware.android.ordered;

import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.support.v4.app.NotificationCompat;

public class NoticeReceiver extends BroadcastReceiver {
    private static final int NOTIFY_ME_ID=1337;

    @Override
    public void onReceive(Context ctxt, Intent intent) {
        NotificationManager mgr=
            (NotificationManager)ctxt.getSystemService(Context.NOTIFICATION_SERVICE);
        NotificationCompat.Builder b=new NotificationCompat.Builder(ctxt);
        PendingIntent pi=
            PendingIntent.getActivity(ctxt, 0,
                new Intent(ctxt, OrderedActivity.class), 0);

        b.setAutoCancel(true).setDefaults(Notification.DEFAULT_ALL)
            .setWhen(System.currentTimeMillis())
            .setContentTitle(ctxt.getString(R.string.notify_title))
```
So, if we leave the activity, our alarms are still going off, but we display a Notification instead of updating the Button caption. Our service is oblivious to whether the broadcast is handled by the activity, the manifest-registered BroadcastReceiver, or is totally ignored.

The downside of this particular technique is that ordered broadcasts are ordinary broadcasts in terms of visibility and scope. Each broadcast involves a bit of IPC, and there is the risk that you might create an insecure broadcast, one that other apps can listen to. A better approach to the Activity-or-Notification pattern is to use an event bus, where the communications can be kept purely within your own process. This is covered elsewhere in this book.

Other Scenarios

You might use an ordered broadcast for plugins to your app. Several plugins might handle the broadcast, and which plugin handles which subset of your broadcasts is determined in large part by which plugins the user elected to install. So, you send an ordered broadcast and allow the plugins to use priorities to establish the “pecking order” and handle their particular broadcasts (aborting those they handle, letting the rest pass).

The SMS subsystem in Android uses ordered broadcasts, to allow replacement SMS clients to handle messages, replacing the built-in client. We will examine this in greater detail later in this book.

Big (and Rich) Notifications

Android 4.1 (a.k.a., Jelly Bean) introduced new Notification styles that automatically expand into a “big” area when they are the top Notification in the drawer. These expanded 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:

![BigText Notification](image)

*Figure 215: BigText Notification, from NotificationCompat Sample App, on a Nexus S*

We also have the Inbox style, which is the same basic concept but designed, for several discrete lines of text:
And, we have the BigPicture style, ideal for a photo, album cover, or the like:
The Builders

Notification.Builder, from the Android SDK, has 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 v10 version of the Android Support package has a version of NotificationCompat that supports these new APIs. Note, though, that older versions of the package do not. You will either need to use a v10 or higher version of
the package or use Jake Wharton's NotificationCompat2, which contains the missing functionality.

The Sample

To see the new Jelly Bean capabilities in action, 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 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.

In the libs/ directory, we have a copy of the v10 version of the Android Support package's android-support-v4.jar.

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:

```java
private NotificationCompat.Builder buildNormal() {
    NotificationCompat.Builder b = new NotificationCompat.Builder(this);

    b.setAutoCancel(true)
        .setDefaults(Notification.DEFAULT_ALL)
        .setWhen(System.currentTimeMillis())
        .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(NotificationCompat.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 wraps our desired Intent action string (here, Settings.ACTION_SETTINGS) in an Intent:

```java
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:

```java
@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`.

**The Results**

If we run our app, we get this:

![Figure 218: Expanded Notification in Drawer, on a Nexus S](image)

From top to bottom, we have:

- Our content text
- Our appended lines of text
NOTIFICATIONS

- 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-Jelly Bean device, the appearance is the normal style, as defined by our buildNormal() method, though on Jelly Bean devices the user can use a two-finger downward swipe gesture to expand the un-expanded Notification.

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”.

697
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

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](https://commonsware.com) sample project is a clone of the [Notifications/DownloadNotify](https://commonsware.com) 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:

```java
startForeground(FOREGROUND_ID,
    buildForegroundNotification(filename));
```

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:

```java
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()`:

```java
stopForeground(true);
raiseNotification(i, null, e2);
```

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.

`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. Users visiting an app's page in Settings will see a “Show notifications” checkbox:

![Figure 219: Show Notifications Checkbox, on a Nexus S](image)

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, at least through 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.
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.

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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book's GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.

Step #1: Adding the InstallReceiver

The reason we used an ordered broadcast (sendOrderedBroadcast()) back in Tutorial #16 for broadcasting the install-completed event is to support this tutorial. Here, we are using the ordered broadcast event pattern:

1. Implement a high-priority receiver in the foreground activity, which handles the event and aborts the broadcast
2. Implement a standard-priority receiver that is registered via the manifest, and have it handle the event for cases where the activity is not in the foreground
So, we need another BroadcastReceiver — let’s create one named InstallReceiver for this role. We will also take advantage of the fact that this broadcast is only needed internally within our app, so we will mark this BroadcastReceiver as non-exported in the manifest, so no other code will be able to trigger it.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

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 InstallReceiver 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 InstallReceiver 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 InstallReceiver, following the same approach that we used for other receivers in this application. However, in addition to choosing InstallReceiver as the component name, also switch the “Exported” drop-down to false.

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 InstallReceiver 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, type in com.commonsware.empublite.action.UPDATE_READY, since this is a custom action and therefore will not appear in the Eclipse drop-down list

Outside of Eclipse

Create an empty src/com/commonsware/empublite/InstallReceiver.java source file; we will fill in the source code for it in the next step.
Also, add the following `<receiver>` element as a child of the `<application>` element in AndroidManifest.xml:

```xml
<receiver
    android:name="InstallReceiver"
    android:exported="false">
    <intent-filter>
        <action android:name="com.commonsware.empublite.action.UPDATE_READY"/>
    </intent-filter>
</receiver>
```

**Step #2: Completing the InstallReceiver**

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!**

Then, modify the InstallReceiver implementation from the original stub to this:

```java
package com.commonsware.empublite;

import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.support.v4.app.NotificationCompat;

public class InstallReceiver extends BroadcastReceiver {
    private static final int NOTIFY_ID=1337;

    @Override
    public void onReceive(Context ctxt, Intent i) {
        NotificationCompat.Builder builder =
            new NotificationCompat.Builder(ctxt);
        Intent toLaunch=
            new Intent(ctxt, EmPubLiteActivity.class);
        PendingIntent pi=
            PendingIntent.getActivity(ctxt, 0, toLaunch, 0);

        builder.setAutoCancel(true).setContentIntent(pi)
            .setContentTitle(ctxt.getString(R.string.update_complete))
            .setContentText(ctxt.getString(R.string.update_desc))
            .setSmallIcon(android.R.drawable.stat_sys_download_done)
            .setTicker(ctxt.getString(R.string.update_complete))
            .setWhen(System.currentTimeMillis());

        NotificationManager mgr=
```
Here, we:

- Create a NotificationCompat.Builder
- Create an activity PendingIntent, pointing at EmPubLiteActivity
- Configure the Notification via the Builder
- Raise the Notification once configured

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.
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., the Galaxy Note is ~5” and from a design standpoint tends to be thought of as a tablet, even though it has telephony capability).

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
LARGE-SCREEN STRATEGIES AND TACTICS

which the user can attempt to become the next Rembrandt, Picasso, or Pollock. The 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:

![Gmail, On a Galaxy Nexus, Showing Conversations](image)

... and the list of labels on another screen:
... and the list of messages in some selected conversation in a third screen:
Figure 222: 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:
Figure 223: 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:
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:
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
<?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>
```
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:

```java
getSupportFragmentManager().beginTransaction()
    .add(R.id.countries, someFragmentHere)
    .commit();
```

In principle, you could have a res/layout-w720dp/main.xml that holds both of the same FrameLayout containers, but just in a vertical LinearLayout:

```xml
<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
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/countries"
    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>
```
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, typically in a separate layout managed by a separate activity.

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:

```java
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
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):

```java
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:

```java
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,
```
```java
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));
```
CountriesFragment

The fragment responsible for rendering the list of EU nations is CountriesFragment. It is a SherlockListFragment, using a CountryAdapter to populate the list:

```java
class CountryAdapter extends ArrayAdapter
{
    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, null);
        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.

To accomplish this, we use the contract pattern, seen in a previous chapter. The fragment defines an interface, which is the “contract” that all hosting activities of that fragment must implement. As before, this requirement is enforced by the superclass, ContractListFragment:
/***/

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 com.actionbarsherlock.app.SherlockListFragment;

public class ContractListFragment<T> extends SherlockListFragment {
  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);
  }
}
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:

```java
@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:

```java
package com.commonsware.android.eu4you;

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
<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
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/countries"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

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:

```java
package com.commonsware.android.eu4you;

import android.content.Intent;
import android.os.Bundle;
```
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class EU4You extends SherlockFragmentActivity 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) getSupportFragmentManager().findFragmentById(R.id.countries);

        if (countries == null) {
            countries = new CountriesFragment();
            getSupportFragmentManager().beginTransaction().add(R.id.countries, countries).commit();
        }

        details = (DetailsFragment) getSupportFragmentManager().findFragmentById(R.id.details);

        if (details == null && findViewById(R.id.details) != null) {
            details = new DetailsFragment();
            getSupportFragmentManager().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);
            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.os.Bundle;
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class DetailsActivity extends SherlockFragmentActivity {

    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=
```
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:
Figure 226: EU4You, On a Tablet Emulator, Landscape

Tapping on a country brings up the details on the right:
In any other configuration, such as a smaller-screen device, we only see the CountriesFragment at the outset:
Figure 228: EU4You, On a Phone Emulator

Tapping on a country brings up the DetailsFragment full-screen in the DetailsActivity:
Other European Flavors

The EU4You sample from above is one way of approaching this master-detail pattern. It is not the only one. In this section, we will review two more implementations of EU4You, one that uses a static fragment for the countries, and one that uses a single activity rather than two.

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
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
```
Similarly, our dual-pane layout uses a `<fragment>` element for the CountriesFragment, alongside the FrameLayout for the details:

```xml
<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:

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    details = (DetailsFragment) getSupportFragmentManager().findFragmentById(R.id.details);

    if (details == null && findViewById(R.id.details) != null) {
        details = new DetailsFragment();
        getSupportFragmentManager().beginTransaction()
            .add(R.id.details, details).commit();
    }
}
```
Neither CountriesFragment or anything involving the details necessarily needs to change.

**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:

```xml
<?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:

```java
@override
public void onCountrySelected(Country c) {
    String url = getString(c.url);
    details.loadUrl(url);
    if (details.getId() != R.id.details) {
        getSupportFragmentManager().beginTransaction().replace(R.id.mainfrag, details, TAGDETAILS).
            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 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:

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    countries = (CountriesFragment) getSupportFragmentManager().findFragmentByTag(TAG_COUNTRIES);
    details = (DetailsFragment) getSupportFragmentManager().findFragmentByTag(TAGDETAILS);

    if (countries == null) {
        countries = new CountriesFragment();
        getSupportFragmentManager().beginTransaction()
            .add(R.id.mainfrag, countries, TAG_COUNTRIES).commit();
    }

    if (details == null) {
        details = new DetailsFragment();

        if (findViewById(R.id.details) != null) {
            getSupportFragmentManager().beginTransaction()
                .add(R.id.details, details, TAGDETAILS).commit();
        }
    } else {
        if (details.getId() == R.id.mainfrag) {
            if (findViewById(R.id.details) != null) {
                getSupportFragmentManager().popBackStackImmediate();
            }
        } else {
            getSupportFragmentManager().beginTransaction().remove(details)
                .commit();
        }
    }
}
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:
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:

```java
package com.commonsware.android.eu4you2;

import android.os.Bundle;

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) {
      // ...
    }
  }
}
```
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.

The SlidingPaneLayout Variant

The Rev13 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 Java Docs 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:

- 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
<?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"/>
```
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 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:

```java
@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:

```java
@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.os.Bundle;
import android.support.v4.widget.SlidingPaneLayout;
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class EU4You extends SherlockFragmentActivity implements CountriesFragment.Contract {
    private DetailsFragment details = null;
    private SlidingPaneLayout panes = null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        details = (DetailsFragment) getSupportFragmentManager().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:

• Our master perspective has a thin strip on the right, showing a peek of the detail fragment.

![Figure 230: EU4You Sliding Pane, On a Phone Emulator, Showing Master](image)

• 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 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: getPageSize(). 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:

```xml
<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:

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    pager=(ViewPager)findViewById(R.id.pager);
}
```
In this case, SampleAdapter is not a FragmentPagerAdapter, nor a FragmentStatePagerAdapter. Instead, it is its own implementation of the PagerAdapter interface:

```java
/*
 * Inspired by
 * https://gist.github.com/8cbe094bb7a783e37ad1
 */
private class SampleAdapter extends PagerAdapter {
    @Override
    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;
    }
}
```

In this case, SampleAdapter is not a FragmentPagerAdapter, nor a FragmentStatePagerAdapter. Instead, it is its own implementation of the PagerAdapter interface:

```java
/*
 * Inspired by
 * https://gist.github.com/8cbe094bb7a783e37ad1
 */
private class SampleAdapter extends PagerAdapter {
    @Override
    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;
    }
}
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 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 page, 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:
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).
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 page. Ideally, ViewPager would subtract out the page margin \textit{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.

**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 gameplay, for example. Applications with other forms of specialized user interfaces — painting apps, photo editors, etc. – may also be better served by eschewing fragments \textit{for those specific activities} and doing something else.
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 Google TV?**

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 is also Google TV to consider, as it presents itself as a -large (720p) or -xlarge (1080p) screen. Fragments can certainly help with displaying a UI for Google 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”).

More coverage of developing for Google TV can be found in a later chapter of this 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 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 dp), 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
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.

**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
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.
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.
<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:

```xml
<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.

Note that <compatible-screens> was added in API Level 9, but that simply means that your build target will need to be API Level 9 or higher. Since <compatible-screens> only affects markets, not your app's runtime behavior, there is no harm in having this element in your manifest when it is run on older devices.

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:

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

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>
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 Google TV, and is the density used for 720p Google 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.

The new wave of 1080p phones are -xxhdpi, 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. Also, while 1080p phones were few in number in early 2013, it is likely to be a popular form factor and will become significant in numbers over the next couple of years.

Also, bear in mind that the Nexus 10, for inexplicable reasons, uses -xxhdpi resources for *launcher icons*, even though the device itself is -xhdpi for everything else.
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 -large and -xlarge 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 -large and -xlarge devices, 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.

Note that if you are importing the previous code to begin work here, you will also need the copy of ActionBarSherlock in this book’s GitHub repository, and to make sure that your imported EmPubLite project references the ActionBarSherlock project as a library.

**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 -large devices, with customized versions of main.xml to be
used by EmPubLiteActivity. Android will automatically use -large resources on -xlarge devices if -xlarge equivalents do not exist.

NOTE: This tutorial has you use the legacy -large resource set qualifiers, instead of -swNNNdp for some value of NNN. There is nothing wrong with using -large in modern development, and in fact it is necessary to support the smattering of pre-API Level 13 tablets. That being said, a future edition of this tutorial will switch to showing you -sw600dp, along with using layout aliases for older device support, as described in the chapter on resources and configuration changes.

If you wish to make this change using Eclipse’s wizards and tools, follow the instructions in the “Eclipse” section below. Otherwise, follow the instructions in the “Outside of Eclipse” section (appears after the “Eclipse” section).

**Eclipse**

First, right-click over the res/ folder, and choose New > Folder from the context menu. Fill in layout-large-land as the folder name, then click “Finish” to create the folder.

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-large-land/ folder and choose “Paste” from the context menu. This makes a copy of your main.xml resource that we can use for -large-land devices.

Double-click on the res/layout-large-land/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. Switch over to the XML editor and fill in 0dp for android:layout_width for the RelativeLayout (this cannot be done in the Properties pane due to a bug in the current version of the tools).

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. Make the following adjustments to the properties of the View using the Properties pane:
TUTORIAL #19 - SUPPORTING LARGEScreens

- Set the Id to @+id/divider
- Set the Height to match_parent
- Set the Background to #AA000000
- Set the Visibility to gone

Then, switch over to the XML and give the View an android:layout_width of 2dp. Also, 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

Then, switch over to the XML and give the FrameLayout an android:layout_width of 0dp.

Save your changes (e.g., <Ctrl>-<S>).

Then, right-click over the res/ folder, and choose New > Folder from the context menu. Fill in layout-large as the folder name, then click “Finish” to create the folder.

Then, right-click over the res/layout-large-land/main.xml file and choose “Copy” from the context menu. After that, right-click over the new res/layout-large/ folder and choose “Paste” from the context menu.

Double-click on res/layout-large/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>).

---

755
Outside of Eclipse

Create a res/layout-large-land/ directory in your project, and create a main.xml file in there with the following contents:

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <RelativeLayout
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="7">
        <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="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"
        android:layout_weight="0"/>
</LinearLayout>
</LinearLayout>
```

Then, create a res/layout-large/ directory in your project, and create a main.xml file in there with the following contents:
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/foo"
    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">

        <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"/>

    <FrameLayout
        android:id="@+id/sidebar"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="0">

    </FrameLayout>

</LinearLayout>

**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:
private View sidebar=null;
private View divider=null;

Then, in `onCreate()` of `EmPubLiteActivity`, initialize those data members, sometime after the call to `setContentView()`:

```java
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`:

```java
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
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:

```java
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:

```java
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:

```java
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:

```java
help = (SimpleContentFragment)getSupportFragmentManager().findFragmentByTag(HELP);
about = (SimpleContentFragment)getSupportFragmentManager().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:

```java
void showAbout() {
    if (sidebar != null) {
```
Both of these methods follow 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
• 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:

```java
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    case android.R.id.home:
        pager.setCurrentItem(0, false);
        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);
    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);
    }
    return(super.onOptionsItemSelected(item));
}
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 lines, sometime after you initialized the sidebar and divider data members:

```java
getSupportFragmentManager().addOnBackStackChangedListener(this);
if (getSupportFragmentManager().getBackStackEntryCount() > 0) {
    openSidebar();
}
```

The first statement registers our activity as receiving events related to changes in the state of the back stack. The rest of that code will reopen our sidebar if, due to a configuration change, we have fragments on the back stack — by default, our sidebar is closed, as that is the state that is encoded in the layout files.

To make this compile, we need to implement onBackStackChanged() in EmPubLiteActivity:

```java
@Override
public void onBackStackChanged() {
    if (getSupportFragmentManager().getBackStackEntryCount() == 0) {
        LinearLayout.LayoutParams p =
            (LinearLayout.LayoutParams)sidebar.getLayoutParams();
        if (p.weight > 0) {
            p.weight = 0;
            sidebar.setLayoutParams(p);
            divider.setVisibility(View.GONE);
        }
    }
}
```

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.

At this point, if you build the project and run it on a -large or -xlarge device or emulator (e.g., a WXGA800 emulator image with default settings), and you choose to view the notes, help, or about, you will see the sidebar appear, whether in portrait or landscape.
The War of the Worlds

by H. G. Wells [1898]

But who shall dwell in those worlds if they be inhabitable? ... Are we or they lords of the world? ... And how are all things made for man...

HELP (quoted in The Anatomy of Melancholy)

Figure 232: EmPubLight, XLarge Landscape, With Help
Backwards Compatibility Strategies
and Tactics

Android is an ever-moving target. The first Android device (T-Mobile G1/HTC Dream) was released in October 2008, running Android 1.0. In December 2011, the Galaxy Nexus was released, running Android 4.0. Hence, we have averaged one major release per year, plus numerous significant minor releases (e.g., 2.1, 2.2, 2.3).

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 people 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. 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”.

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 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., Barnes & Noble NOOK series), 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 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.

There is an android:maxSdkVersion, which indicates the newest version of Android you would like to support. However, this will only serve as a filter on the Play Store. If a user has, say, a Gingerbread device, and your app has android:maxSdkVersion="10", and the user’s device gets an upgrade to Ice Cream Sandwich, your app may remain installed. In that case, your app will be running on a version higher than the maximum you specified. However, you will not show up in the Market for devices running a newer version of Android than you specified. Google strongly discourages the use of this attribute.

The fourth place — and perhaps the one that confuses developers the most – is the build target.

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 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.

We have seen two of these so far in the book:

- The Android Support package, offering implementations of fragments and loaders going back to Android 1.6
- ActionBarSherlock, providing Android 2.x devices (and beyond) with action bars
In these cases, the API for using the compatibility library is nearly identical to using the native Android capability, mostly involving slightly differentpackage 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. For example, Android 4.0 added a GridLayout, to try to simplify some UI patterns that are tedious to do with nested LinearLayout containers or a RelativeLayout. While GridLayout itself is only available natively on Android starting with 4.0, it is entirely possible to take the source code for GridLayout and get it working on older devices, as one developer did. Of course, after that developer went through all of that work, Google added GridLayout to the Android Support package.

These sorts of backports can then be dropped once you drop support for the older devices that required them. For example, if the action bar APIs stay stable, ActionBarSherlock will no longer be needed once you drop support for Android 2.x devices, perhaps sometime late in 2013.

**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.

**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:

```java
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, in the tutorials, we used a WebViewFragment back-ported to work with the Android Support version of fragments and ActionBarSherlock. WebViewFragment wants to pass the onResume() and onPause() events to the WebView it manages, but onResume() and onPause() only exist on WebView on API Level 11 and higher. So, we need to use the Build version guard to ensure we do not call those methods on older devices. To get rid of the warning messages, we use @TargetAPI(11):

```java
/**
 * Called when the fragment is visible to the user and actively running. Resumes the WebView.
 */
@TargetApi(11)
@Override
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)
```
Now, Lint knows that we are intentionally using API Level 11 capabilities and will no longer warn us about them.

**Another Example: AsyncTask**

As mentioned in the chapter on threads, AsyncTask can work with either a full thread pool or a “serialized executor” that will only execute one AsyncTask at a time. From Android 1.6 through 2.3, the full thread pool is the only available option. Android 3.0 introduced the serialized executor, and Android 3.2 made it the default, if you have set your targetSdkVersion to be 13 or higher.

If you want to ensure that no matter what your targetSdkVersion is, that you always get the full thread pool, you need to use a version guard block:

```java
@TargetApi(11)
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);
    }
}
```

Here, we use executeOnExecutor() and specifically request the THREAD_POOL_EXECUTOR — but only on API Level 11 and higher. Otherwise, we fall back to the default behavior, which gives us the thread pool used on the older API levels.

**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.

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
<?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.

**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 4.2/API Level 17](https://developer.android.com/studio/releases).

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](https://developer.android.com/studio/releases) points out the changes between API Level 16 and API Level 17.

Other changes are called out in [the JavaDocs for Build.VERSION_CODES](https://developer.android.com/studio/releases), 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.
BACKWARDS COMPATIBILITY STRATEGIES AND TACTICS

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.
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, bear in mind that tutorials written before late August 2008 are probably written for the M5 SDK and, as such, will require considerable adjustment to work properly in current SDKs.

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. StackOverflow’s android tag
2. android-developers, for SDK questions and answers
3. adt-dev, for questions and answers about the official Android development tools
4. android-discuss, designed for free-form discussion of anything Android-related, not necessarily for programming questions and answers

You might also consider:

1. The core Android team’s periodic Hangouts on Google+
2. The Android tutorials and programming forums over at anddev.org
3. The #android-dev IRC channel on freenode (irc.freenode.net)

It is important, particularly for StackOverflow 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)
2. The stack trace from LogCat, if the problem is an unhandled exception
3. On StackOverflow, 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
4. 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)
5. On StackOverflow, 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
6. 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

Note that, as of the time of this writing, you cannot browse or search the Android source code from the Android project’s site. The easiest way to browse the source
code is to browse the GitHub mirrors of the source. To search the source code, you can use services like AndroidXRef.

**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.
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 the Eclipse graphical layout editor. 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

Once upon a time, most layouts were implemented using a combination of LinearLayout, RelativeLayout, and TableLayout. In fact, most layouts are still created using those three “classic” containers. 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 Eclipse’s 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:

- 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.

Eclipse Drag-and-Drop

Where RelativeLayout falls down is with the drag-and-drop capability of the graphical layout editor in Eclipse.

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 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 TableRow, 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 TableRow, 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 TableRow 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 TableRow 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
Eclipse is beyond hopeless with GridLayout at this junction. With luck, it will improve in future years. In the interim, you will most likely need to spend quality time in the XML editor in order to get the results that you want.

**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 android-support-v4.jar, as GridLayout requires some resources. Hence, it is in an Android library project that you must add to your project.

You will find this library project in $SDK/extras/android/support/v7/gridlayout, where $SDK is wherever you installed your copy of the Android SDK.

Command-line builds can use `android update lib-project` to attach the Android library project to their host project. Eclipse developers will need to create a new Eclipse project based upon the Android library project first. In either case, the process is similar to what was needed to add ActionBarSherlock to a project, as was described in one of the tutorials.

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/your.package.goes.here, where your.package.goes.here is replaced by your application's package name. If you use Eclipse to add the GridLayout to the layout resource, it will automatically add this namespace, under the prefix of app, such as:

```xml
<android.support.v7.widget.GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    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.

**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`.

In some future edition of this book, when Eclipse editing of `GridLayout` is sensible, we will describe the process of using it to create the various layouts that appear in the rest of this chapter.

**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.

![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.

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:

```java
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:

```java
package com.commonsware.android.gridlayout;

import android.content.Context;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;
import java.util.ArrayList;

public class SampleAdapter extends FragmentPagerAdapter {
```
public 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));
}

public SampleAdapter(Context ctx, FragmentManager mgr) {
    super(mgr);
    this.ctxt = ctx;
}

@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:
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) {
    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 not only for ActionBarSherlock, but also 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:layout_width="match_parent"
    android:layout_height="match_parent"
    app:columnCount="2">
```
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 234: Row Using GridLayout, on a 4.0.3 Emulator
INTRODUCING GRID LAYOUT

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:

```xml
    android:layout_width="match_parent"
    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>
```
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.

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`:

```xml
<android.support.v7.widget.GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:columnCount="2">
  <TextView
    app:layout_column="0"/>
  <EditText
    app:layout_column="1"/>
  <TextView
    app:layout_column="0"/>
  <EditText
    app:layout_column="1"/>
</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:

![Figure 236: “Show Structure” Toolbar Icon](image)

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:
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`:

```xml
<android.support.v7.widget.GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    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"
        app:layout_gravity="fill_horizontal"
        android:inputType="textPersonName"
        android:requestFocus/>

    <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:

![Table Using GridLayout, on a 4.0.3 Emulator](image)

That holds true regardless of how wide that column is:
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:

```xml
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```
INTRODUCING GRIDLAYOUT

```xml
<android.support.v7.widget.GridLayout
    app:columnCount="2"
    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">
        <requestFocus/>
    </EditText>
</android.support.v7.widget.GridLayout>
```

Visually, this sample is identical to the last one:

![GridLayout Sampler](image)

*Figure 240: Table Using GridLayout and Implicit Positions, on a 4.0.3 Emulator*
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 241: Span Sample (image courtesy of Android Open Source Project)](image)

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.
A GridLayout-based layout that implements the above diagram can be found in res/layout/spans.xml:

```
    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"/>

  <android.support.v7.widget.Space
      android:layout_width="36dp"
      app:layout_column="1"/>
```

INTRODUCING GRIDLAYOUT

798
INTRODUCING GRIDLAYOUT

```xml
<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_width="36dp"
    app:layout_column="8"
    app:layout_row="0"/>

<android.support.v7.widget.Space
    android:layout_width="36dp"
    app:layout_column="8"
    app:layout_row="1"/>

<android.support.v7.widget.Space
    android:layout_width="36dp"
    app:layout_column="8"
    app:layout_row="2"/>

<android.support.v7.widget.Space
    android:layout_width="36dp"
    app:layout_column="8"
    app:layout_row="3"/>

<android.support.v7.widget.Space
    android:layout_height="36dp"
```
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:

![GridLayout Sampler](image)

*Figure 242: GridLayout Spans, on a 4.0.3 Emulator*

However, the fixed-sized Space elements break the fluidity of the layout:
Perhaps someday someone will create a \texttt{PercentSpace} widget, occupying a percentage of the parent's size, that could be used instead.

The author would like to give thanks to \url{those on StackOverflow who assisted in getting the span layout to work}.

**Should You Use GridLayout?**

An \url{Android Developers Blog post on GridLayout} says:

If you are starting a UI from scratch and are not familiar with Android layouts, use a GridLayout — it supports most of the features of the other layouts and has a simpler and more general API than either TableLayout or RelativeLayout.

In 2014, that may be a sound recommendation for newcomers to Android, and it might not be an unreasonable suggestion for experienced Android developers today. However, the complexity of the Android library project required for the backport, coupled with the copious number of examples using the classic layout managers, make those older layout managers a better choice in many respects for those with limited Android experience.
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.
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 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
<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:

```java
package com.commonsware.android.chrono;
import android.app.Activity;
import android.app.DatePickerDialog;
import android.app.TimePickerDialog;
```
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 dateTimeLabel;
    Calendar dateAndTime = Calendar.getInstance();

    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);

        dateTimeLabel = (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() {
        dateTimeLabel.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();
        }
    };
}

805
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:
Figure 244: ChronoDemo, As Initially Launched, on Android 4.0.3

Figure 245: ChronoDemo, Showing DatePickerDialog
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:
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 StackOverflow, 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.
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
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:

```java
package com.commonsware.android.dlgfrag;

import android.app.AlertDialog;
import android.app.Dialog;
import android.content.DialogInterface;
import android.os.Bundle;
import android.support.v4.app.DialogFragment;
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!");
    }
}
```
In `onCreateDialog()`, we inflate a custom layout (R.layout.dialog) that consists of some `TextView` labels and `EditText` fields:

```xml
<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>
</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`:

```java
public void showMe(View v) {
    new SampleDialogFragment().show(getSupportFragmentManager(),
                                    "sample");
}
```
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:

![Dialog Fragment Demo](image)

*Figure 248: SampleDialogFragment, As InitiallyLaunched, 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.
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.
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:
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 nonsense words broken down into five groups of five, as seen in the HeaderDetailList activity:

```java
private static final String[][] items = {
    { "lorem", "ipsum", "dolor", "sit", "amet" },
```

*Figure 249: ListView with Row Structure Mix (image courtesy of Google)*
**ADVANCED LIST VIEWS**

```
{ "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:

![Batch #1](image1)

![Batch #2](image2)

*Figure 250: 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:
@Override
public int getCount() {
    int count = 0;
    for (String[] batch : items) {
        count += 1 + batch.length;
    }
    return(count);
}

- getItemCount() 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 nonsense 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) {

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:

  ```java
  @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:

  ```java
  @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
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()`:

```java
@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);
```
### Advanced ListViews

```java
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 nonsense 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:

```java
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.
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
<?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
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated">
  </style>
</resources>
```

- Use that style as the background of your ListView (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
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 nonsense words. In this case, you can rate the words on a three-star rating. Words given a top rating are put in all caps:

```java
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
    // More code...
}
```
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);
    }
}
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 nonsense 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 nonsense 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.
ADVANCED LISTVIEWS

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
<?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:
Figure 251: RateList, As Initially Shown

This includes the toggled rating bars turning their words into all caps:

Figure 252: RateList, With a Three-Star Word
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:

```java
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());
  }

  private View buildHeader() {
    // Create and return the header View
  }
}
```
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);
    }
}
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 253: A ListView with a header view shown](image)

If you scroll downward, the header will slide off the top, and eventually the footer will scroll into view:
Figure 254: A ListView with a footer view shown
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.

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.

To set this up:
ACTION BAR NAVIGATION

1. Call `setNavigationMode(ActionBar.NAVIGATION_MODE_LIST)` on the `ActionBar` to enable the list navigation mode, which you get via `getActionBar()` (or `getSupportActionBar()` for `ActionBarSherlock` apps).

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/ListNav` 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:

```java
package com.commonsware.android.listnav;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;
import com.actionbarsherlock.app.SherlockFragment;

public class EditorFragment extends SherlockFragment {
    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);
    }

    void setHint(CharSequence hint) {
```
Setting up the list navigation mode is part of the work we do in `onCreate()`:

```java
ArrayAdapter<String> nav = null;
ActionBar bar = getSupportActionBar();

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 (Ice Cream Sandwich) offers 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:

- We set up our `EditorFragment` in `onCreate()`, if it does not already exist:
ACTION BAR NAVIGATION

frag = (EditorFragment) getSupportFragmentManager().findFragmentById(android.R.id.content);

if (frag == null) {
    frag = new EditorFragment();
    getSupportFragmentManager().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()`:

```java
@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, that we are going to use 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 `FragmentManager`s. Or, we could have a variety of distinct fragment instances, from different classes, and swap between them using `FragmentManager`s. What you do to update your UI based upon the list navigation change is up to you.
ACTION BAR NAVIGATION

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:

In onCreate(), we restore our models array:

```java
if (state != null) {
    models = state.getCharSequenceArray(KEY_MODELS);
}
```

And, later in onCreate(), we tell the action bar which position to select:

```java
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:
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()` (or `getSupportActionBar()` for ActionBarSherlock apps)
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/TabFragmentDemo sample project is very similar to the one for list navigation described above, except that it uses tabs instead of list navigation. We 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:

```java
package com.commonsware.android.tabfrag;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;
import com.actionbarsherlock.app.SherlockFragment;

public class EditorFragment extends SherlockFragment {
    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);
    }
}
```
In `onCreate()`, we tell the ActionBar that we want tab navigation, then we add 10 tabs to the bar:

```
ActionBar bar = getSupportActionBar();
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:

```java
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) getSupportFragmentManager().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:
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:
Figure 257: TabFragmentDemo, on Android 4.0.3, Phone-Sized Screen in Landscape

or on a large-sized screen in portrait:

Figure 258: TabFragmentDemo, on Android 4.0.3, Tablet-Sized Screen in Portrait
ACTION BAR NAVIGATION

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.
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 259: The Gmail action bar on a Honeycomb tablet, in normal mode](image)

...to one designed to perform operations on what you have selected:

![Figure 260: The Gmail action bar on a Honeycomb tablet, showing an action mode](image)

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 both action modes and context menus.
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.
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 bar takes advantage of the fact that we have a lot more screen space on a tablet than we do on a phone, and uses some of that space to consistently benefit the user.

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/Manual sample project. This is another variation on the “show a list of nonsense 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:
### Action Modes and Context Menus

- 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)

In the case of the ActionMode 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:

```java
@Override
public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    initAdapter();
    getListView().setLongClickable(true);
    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
    getListView().setOnItemLongClickListener(new ActionModeHelper(this, this));
}
```

### 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:

```java
@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 the “activated” style:

```xml
<?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>
```

That style is defined for Honeycomb and higher in `res/values-v11/styles.xml`:

```xml
<?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>
```

A do-nothing version of that style is used for older devices, from `res/values/styles.xml`:

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="activated"/>
</resources>
```
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:

```java
@Override
public boolean onCreateActionMode(ActionMode mode, Menu menu) {
    MenuInflater inflater = host.getSupportMenuInflater();

    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 261: The ActionMode sample application's action bar on a Honeycomb tablet, showing the active action mode*

**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:

```java
@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:

```java
@SuppressWarnings("unchecked")
public boolean performAction(int itemId, int position) {
    ArrayAdapter<String> adapter=(ArrayAdapter<String>)getListAdapter();

    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);
}
```

And, the onActionItemClicked() method calls performAction():

ACTION MODES AND CONTEXT MENUS
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:

```java
@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-Modal-Choice 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 operations on the checked items. For this scenario, Android has a new built-in
**ACTION MODES AND CONTEXT MENUS**

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 Honeycomb. More importantly, though, this version of the sample uses the native API Level 11+ version of the action bar, as ActionBarSherlock does not support CHOICE_MODE_MULTIPLE_MODAL at this time.

Once again, in onCreate(), we need to set up the smarts for our ListView. This time, though, we will use CHOICE_MODE_MULTIPLE_MODAL:

```java
@TargetApi(11)
@Override
public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    initAdapter();

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        getListView().setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
        getListView().setMultiChoiceModeListener(new HCMultiChoiceModeListener(this,
            getListView()));
    }
    else {
        getListView().setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);
        registerForContextMenu(getListView());
    }
}
```

If we are on an API Level 11+ device, 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.

We will discuss the non-Honeycomb branch later in this chapter.

Since we now may have multiple checked items, our performAction() method must take this into account, capitalizing or removing all checked words:

```java
@ SuppressWarnings("unchecked")
public boolean performActions(MenuItem item) {
    ArrayAdapter<String> adapter=(ArrayAdapter<String>)getListAdapter();
}
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.removeCase(word, position);
            }
        }

        return(true);
    
    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.removeCase(words.get(position));
        }

        getListView().clearChoices();

        return(true);

    return(false);
}

Back in the Honeycomb-or-higher code, 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;

856
ACTION MODES AND CONTEXT MENUS

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;
    }

    @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:

![Figure 262: The ActionModeMC sample application's action bar on a Honeycomb tablet, showing the active action mode](image)

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:

```java
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:

```java
@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:

```java
@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:

```java
@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 rows, and the easiest way to do that is to call `setAdapter()` on it, passing in its existing adapter:

```java
@override
public void onDestroyActionMode(ActionMode mode) {
    if (activeMode != null) {
        activeMode=null;
        getListView().setChoiceMode(ListView.CHICK_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 263: Action Mode Long Press Demo, As Initially Launched]

Tapping on a row provides the normal momentary highlight.

However, if the user long-clicks a row, we move into the action mode and a multiple-choice ListView:
ACTION MODES AND CONTEXT MENUS

Figure 264: Action Mode Long Press Demo, with Action Mode Activated

Figure 265: 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:

![Figure 266: The ActionModeMC sample on a phone](image)

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:
If there is more horizontal room (i.e., it is not “narrow”), then the action mode will display as normal:
What Came Before: Context Menus

Since ActionBarSherlock supports manual action modes (if not the more-convenient multiple-choice action modes), you can use action modes going back to Android 2.1, which is probably more than sufficient for your needs.

However, perhaps there are situations where you truly do want a context menu, rather than the contextual action bar. You are certainly welcome to stick with the older approach. In fact, the multiple-choice action mode sample demonstrates supporting both context menus (on pre-Honeycomb devices) and action modes (on Honeycomb and higher).

Creating a Context Menu

First, you need to indicate which widget(s) on your activity have context menus. To do this, call registerForContextMenu() from your activity, supplying the View that is the widget needing a context menu. In the case of the multiple-choice version of ActionModeDemo, we did this in the non-Honeycomb branch, supplying our ListView as the View in question:

```java
registerForContextMenu(getListView());
```
Next, you need to implement `onCreateContextMenu()`, which, among other things, is passed the `View` you supplied in `registerForContextMenu()`. You can use that to determine which menu to build, assuming your activity has more than one.

The `onCreateContextMenu()` method also gets the `ContextMenu` itself and a `ContextMenu.ContextMenuInfo`, which tells you which item in the list the user did the tap-and-hold over, in case you want to customize the context menu based on that information. For example, you could toggle a checkable menu choice based upon the current state of the item.

It is also important to note that `onCreateContextMenu()` gets called for each time the context menu is requested. Unlike the options menu (which is only built once per activity), context menus are discarded once they are used or dismissed. Hence, you do not want to hold onto the supplied `ContextMenu` object; just rely on getting the chance to rebuild the menu to suit your activity’s needs on an on-demand basis based on user actions.

Beyond that, `onCreateContextMenu()` does the same sort of thing as `onCreateOptionsMenu()`: inflate a menu resource to indicate what should appear, such as is the case in `ActionModeDemo`:

```java
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
        ContextMenu.ContextMenuInfo menuInfo) {
    getMenuInflater().inflate(R.menu.context, menu);
}
```

In this case, we are using the same menu resource as is used by the action mode.

**Responding to a Context Menu**

Just as to respond to an action bar item, you implement `onOptionsItemSelected()`; to respond to a context menu item, you implement `onContextItemSelected()`:

```java
@Override
public boolean onContextItemSelected(MenuItem item) {
    boolean result=performActions(item);
    if (!result) {
        result=super.onContextItemSelected(item);
    }
    return(result);
}
```
ACTION MODES AND CONTEXT MENUS

As with onOptionsItemSelected(), you are passed the MenuItem that represents the context menu item that the user chose. In this case, we pass that object to the same performActions() method used by the action mode. If performActions() returns false, we chain to the superclass (in case a built-in context menu item was clicked).
ActionBarCompat: 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. Known as ActionBarCompat, 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.

This section will outline how using ActionBarCompat differs from using ActionBarSherlock, plus reasons why you might choose one over the other.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on the action bar.

Using the ActionBarCompat

The recipe for using the AppCompat action bar is reminiscent of using ActionBarSherlock, and it requires no new skills beyond what you have learned so far in this book. However, there are some subtle and not-so-subtle differences in the approaches each library takes.

To see the differences, we will take a look at the ActionBar/AppCompat sample project. This is a port of the fragments-and-action-bar sample from earlier in the book, where we have replaced ActionBarSherlock with ActionBarCompat.
The Library Project

Just as you need to attach an Android library project for ActionBarSherlock to an app's project to use that backport, so too you must attach an Android library project for ActionBarCompat. This library project can be found in your Android SDK installation, inside the extras/android/support/v7/appcompat directory.

For example, if you are using Eclipse, 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 ActionBarCompat version.

Your Theme

Just as you need to reference a theme that inherits from Theme.Sherlock to use ActionBarSherlock in an activity or application, you must reference a theme that inherits from Theme.AppCompat to use ActionBarCompat.

The original project used Theme.Sherlock in the android:theme attribute of the <application> element in the manifest; this cross-port simply replaces it with Theme.AppCompat:

```xml
<application
    android:allowBackup="false"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.AppCompat">
    ...
</application>
```

You will notice that we no longer have android:uiOptions="splitActionBarWhenNarrow" in <application>, as the original project had. That is intentional, and will be explained a bit later in this section.

Your Menu Resources

Where things start to get a bit strange with ActionBarCompat comes with our menu resources.

ActionBarSherlock works with the same menu resources as we would use with native API Level 11+ action bars. ActionBarCompat does not. Instead, it 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
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/add"
    android:icon="@android:drawable/ic_menu_add"
    android:showAsAction="ifRoom"
    android:title="@string/add" android:actionLayout="@layout/add"/>
  <item
    android:id="@+id/reset"
    android:icon="@android:drawable/ic_menu_revert"
    android:showAsAction="never"
    android:title="@string/reset"/>
</menu>
```

and we had to change it to:

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:thisapp="http://schemas.android.com/apk/res-auto"
  xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/add"
    android:icon="@android:drawable/ic_menu_add"
    android:title="@string/add" thisapp:actionLayout="@layout/add"
    thisapp:showAsAction="ifRoom"/>
  <item
    android:id="@+id/reset"
    android:icon="@android:drawable/ic_menu_revert"
    android:title="@string/reset" thisapp:showAsAction="never"/>
</menu>
```

Note that we have a new `xmlns:thisapp="http://schemas.android.com/apk/res-auto"` namespace declaration in the root `<menu>` element, and that namespace is used for the `thisapp:showAsAction` and `thisapp:actionLayout` attributes. The actual prefix name, here shown as `thisapp`, can be whatever you want — the official documentation samples show it as `yourapp`. It just has to be unique within the document and a valid XML namespace prefix (e.g., no whitespace).

**Your Manifest Metadata**

However, that same use-a-different-namespace trick does not work well with the manifest, as an app does not load up the manifest itself directly, the way we inflate
menu resources. And, while ActionBarSherlock was able to make use of an
android:uiOptions="splitActionBarWhenNarrow" attribute, ActionBarCompat does not.

Instead, we need to use a <meta-data> element on our <activity> to indicate that
we want splitActionBarWhenNarrow behavior:

   <activity
   android:name="ActionBarFragmentActivity"
   android:label="@string/app_name">
   <intent-filter>
      <action android:name="android.intent.action.MAIN"/>
      <category android:name="android.intent.category.LAUNCHER"/>
   </intent-filter>
   <meta-data
      android:name="android.support.UI_OPTIONS"
      android:value="splitActionBarWhenNarrow"/>
</activity>

Your Activity

Just as we had to inherit from a Sherlock-flavored activity class to use
ActionBarSherlock, we have to inherit from an ActionBarActivity class to use
ActionBarCompat. ActionBarActivity itself inherits from FragmentActivity, and
so we can use the Android Support package's backport of fragments without issue.
However, note that there are no other analogues of ActionBarActivity for other
scenarios, whereas ActionBarSherlock offered SherlockListActivity and kin. 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.

Your Fragments

The biggest gain from using ActionBarCompat comes with your fragments. You no
longer need to extend a Sherlock-flavored fragment class. You can use the standard
Android Support package backport version of fragments, like ListFragment.

Your Callback Methods

In many cases, your onCreateOptionsMenu() and onOptionsItemSelected() methods will be the same for ActionBarCompat as they would be for a regular
Android app. You no longer need to use Sherlock-flavored Menu and MenuItem classes, for example.

However, if you do need to manipulate action bar-specific attributes of your inflated menu resources, you will be unable to do so directly. While ActionBarSherlock’s edition of MenuItem has getActionView(), the standard Android one only does for API Level 11+, not earlier versions.

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.

For example, our onCreateOptionsMenu() method needs to configure the EditText that we are adding as an action layout to the action bar. With ActionBarSherlock, we have to use the Sherlock-flavored Menu and MenuItem classes, but we can then work with the action view directly:

```java
private void configureActionItem(Menu menu) {
    EditText add =
        (EditText) menu.findItem(R.id.add).getActionView()
            .findViewById(R.id.title);
    add.setOnEditorActionListener(this);
}
```

With ActionBarCompat, we no longer need the specialized imports, but we do need to use the static getActionView() method on MenuItemCompat to retrieve our inflated View:

```java
private void configureActionItem(Menu menu) {
    EditText add =
        (EditText) MenuItemCompat.getActionView(menu.findItem(R.id.add))
            .findViewById(R.id.title);
    add.setOnEditorActionListener(this);
}
```

Your Results

Visually, the results are nearly identical, as both ActionBarSherlock and ActionBarCompat aim for a consistent look and feel with the native action bar implementation.
One difference comes with the style of the overflow on devices with a MENU button. ActionBarSherlock elected to stick with the Android 2.x options menu style, for consistency with other apps on Android 2.x:

![Image](image_url)

*Figure 269: ActionBarSherlock Overflow, on Android 2.3.3*

ActionBarCompat elects to style the overflow to match the look that debuted with Android 4.x:
Choosing a Backport

You might think that everybody should race to use the ActionBarCompat backport, since it is officially supported by Google. Certainly, you are welcome to do so.

However, if you already have existing ActionBarSherlock-based apps, there is limited utility in replacing ActionBarSherlock with ActionBarCompat. That would require a fair bit of rewriting for little added value.

You should also keep in mind the “track record” of the implementations. While ActionBarCompat is officially supported, it has not been used all that much. The most notable ActionBarCompat action bar app at the time of the backport’s release was the 2013 Google I/O conference app. ActionBarSherlock, on the other hand, has been used by countless apps and therefore has been tried in many more situations. This means that there is a lot more documentation for ActionBarSherlock at this time, from blog posts to StackOverflow questions.
Also, bear in mind that Android 2.x devices are steadily fading away, much as did the 1.x devices that preceded them. Apps begun in 2014 or beyond could easily elect to skip Android 2.x entirely, avoiding the need for any sort of action bar backport.
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.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on the action bar.

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.
OTHER ADVANCED ACTION BAR TECHNIQUES

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:

![SearchView Demo](image)

Figure 271: SearchView Demo, Showing Collapsed Action View

Tapping that opens a search field, taking over more of the action bar:
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
<?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="@android:drawable/ic_menu_search"
        android:showAsAction="ifRoom|collapseActionView"
        android:title="@string/filter"/>
</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.
**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`:

```java
@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`:

```java
private void configureSearchView(Menu menu) {
    MenuItem search = menu.findItem(R.id.search);
    SearchView sv = (SearchView) search.getActionView();
    sv.setQueryTextListener(this);
    sv.setCloseListener(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`
- 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.
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:

```java
@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:

```java
@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.
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:

```java
@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 mollis 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,
ActionBarFragment has a do-nothing implementation of `onQueryTextSubmit()`, simply returning false to indicate that we have not consumed the event:

```java
@override
public boolean onQueryTextSubmit(String query) {
    return(false);
}
```

### `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:

```java
@override
public boolean onClose() {
    adapter.getFilter().filter(null);
    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()` on `MenuItemActionCollapse()` 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:
Figure 273: SearchView Demo, Showing Filtered Results
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.

Of course, the action bar provides other navigation options, like tabs or list navigation. Those may work in many cases. However, currently, tabs are usually thought of as a means of categorizing larger sets of content, with each tab representing a part of the content. In this sense, tabs are “peers”, often with similar...
rendering structures. While list navigation does not have that same sort of semantics, the simple Spinner-style drop-down does not afford much UI flexibility, and longer lists can be difficult to manipulate via the small scrolling area.

Back before the action bar, the “go-to” design pattern for navigation was the so-called “dashboard”:

![Dashboard](image)

*Figure 274: 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:
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.

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:

```xml
<android.support.v4.widget.DrawerLayout
xmlns:android="http://schemas.android.com/apk/res/android"
```
Note that `DrawerLayout` was added in mid-2013, so you will need an `android-support-v4.jar` file that is from that timeframe or newer to have access to this class.

`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.

The ActionBarDrawerToggle

The onCreate() method of MainActivity is responsible for setting up the drawer, as well as the activity’s main content:

```java
@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)findViewByld(R.id.drawer);
    drawer.setAdapter(new ArrayAdapter<String>(
        this,
        R.layout.drawer_row,
        getResources().getStringArray(R.array.drawer_rows)));
    drawer.setOnItemClickListener(this);

    drawerLayout=(DrawerLayout)findViewByld(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:

```java
private void showLorem() {
    if (lorem == null) {
        lorem=new LoremFragment();
    }

    if (!lorem.isVisible()) {
        getFragmentManager().beginTransaction()
            .replace(R.id.content, lorem).commit();
```
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:

```java
@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);
    }
```
IMPLEMENTING A NAVIGATION DRAWER

```java
}

return(super.onOptionsItemSelected(item));
}
```

Without any additional work, launching the app will show the “slice of a hamburger” navigation drawer icon in the action bar:

![Image of app with navigation drawer icon]

*Figure 276: Nav Drawer Sample App, Showing a Bit of Hamburger*

Tapping the app icon will open the drawer:
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.

**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.

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](https://developer.android.com/design/patterns/lists-and-tables) 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
IMPLEMENTING A NAVIGATION DRAWER

Figure 278: 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 279: Gmail Navigation Drawer, with RadioButtons, Dividers, and Badges (Sans Backgrounds)
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:

```xml
<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:choiceMode="singleChoice"/>
</android.support.v4.widget.DrawerLayout>
```
IMPLEMENTING A NAVIGATION DRAWER

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:

```java
@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();
    }
}

@Override
protected void onPostCreate(Bundle savedInstanceState) {
    super.onPostCreate(savedInstanceState);
```
IMPLEMENTING A NAVIGATION DRAWER

Our `showLorem()` method now will `post()` a Runnable, named `onNavChange`, after it calls `commit()` on its `FragmentTransaction`:

```java
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:

```java
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:
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, by inheriting from the same sort of `ContractListFragment` 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:

```java
package com.commonsware.android.drawer.activated;

import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
```
public class LoremFragment extends ContractListFragment>LoremFragment.Contract> {

    @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:
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:

```java
@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.
IMPLEMENTING A NAVIGATION DRAWER

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:

![Image](image_url)

*Figure 282: 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:

```java
private void showContent() {
  if (content == null) {
    content = new ContentFragment();
  }
}
```
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:

```java
@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.
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...
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., drop-down list navigation in the action bar) 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

Implementing a Navigation Drawer
IMPLEMENTING A NAVIGATION DRAWER

- 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.
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.
ADVANCED USES OF WebView

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
<?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:

```java
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 {
```
 ADVANCED USES OF WebView

```java
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 {
```
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 `Locator` 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 `Locator` instance under the name `locater` to the Web content loaded in the WebView

The `Locator` 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 `Locator` 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 `Locator` object would be accessible by JavaScript, including those inherited from superclasses like `Object`. Note that your build target
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.

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
<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.<br/>
</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:
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:
Figure 284: 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 executeJavaScript() counterpart to addJavascriptInterface(), where you could supply some JavaScript source and have it executed within the context of the currently-loaded Web page.

Oddly enough, that is not how this is accomplished.
ADVANCED USES OF WebView

Instead, given your 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:

```java
package com.commonsware.android.geoweb2;

import android.annotation.SuppressLint;
import android.annotation.SuppressLint;
import android.app.Activity;
import android.annotation.TargetApi;
import android.annotation.TargetApi;
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");
    }
```
@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("\n");
        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;
        }
    }
}
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:
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.

NOTE: The behavior documented here is for Android 4.3 and below, and for Android 4.4+ when your app has its android:targetSdkVersion set to 18 or lower. On Android 4.4+, you will want to use the new evaluateJavascript() method to achieve the same effect as had been seen with loadUrl("javascript:..."). This sample will be updated in a future version of the book to accommodate this change.

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())
ADVANCED USES OF WebView

- clearCache() to clear the browser resource cache and clearHistory() to clear the browsing history

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.
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.
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:

![FieldDemo activity with IME overlaying multiple-line EditText](image)

*Figure 285: 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 allowed to input, and this determines the basic set of keys available on the soft keyboard. The available classes are:

1. `text` (the default)
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
<?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"
        />
    </TableRow>
    <TableRow>
        <TextView
            android:text="Multi-line text:"
        />
    </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:

*Figure 286: 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:

![Image of an email address field with the @ symbol on the soft keyboard]

*Figure 287: 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:
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 accessory button

By default, on an EditText where you have specified `android:inputType`, the accessory button 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 accessory button 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 accessory button should look like:

```xml
<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
                android:inputType="text|textEmailAddress"
                android:imeOptions="actionSend"
            />
        </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>
            <TextView
                android:text="Multi-line text:"
            />
            <EditText
                android:inputType="text|textMultiLine|textAutoCorrect"
            />
        </TableRow>
    </TableLayout>
</ScrollView>
```
THE INPUT METHOD FRAMEWORK

Here, we attach a “Send” action to the accessory button 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 accessory button 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 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
The Input Method Framework

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
<?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>
```

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:
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:

```java
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)

This will always close the input method editor. However, bear in mind that there are two ways for a user to have opened that input method editor in the first place:

- If their device does not have a hardware keyboard exposed, and they tap on the EditText, the input method editor should appear
- If they previously dismissed the editor, or if they are using the editor for a widget that does not normally pop one up (e.g., ListView), and they long-tap on the MENU button, the input method editor should appear
The Input Method Framework

If you only want to close the input method editor for the first scenario, but not the second, use `InputMethodManager.HIDE_IMPLICIT_ONLY` as a flag for the second parameter to your call to `hideSoftInputFromWindow()`, instead of the `0` shown in the previous example.
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
<?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">

```

931
Fonts

```<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:"
        android:textSize="20sp"
    />
</TableRow>
```
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:

```java
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(),
```

933
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?
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.

**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.

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.
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”
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.
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 (\textit{\textless b\textgreater}), italics (\textit{\textless i\textgreater}), and underline (\textit{\textless u\textgreater}) are officially supported, such as:

\textit{\textless string name="welcome"}\textgreater Welcome to \textit{\textless b\textgreater}Android\textit{\textless /b\textgreater}!\textit{\textless /string\textgreater}

When you retrieve the string resource via \texttt{getText()}, you get back a `CharSequence` that represents a `Spanned` object with the markup rules in place.
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:

1. `<a href="...">
2. `<b>
3. `<big>
4. `<blockquote>
5. `<br>
6. `<cite>
7. `<dfn>
8. `<div align="...">
9. `<em>
10. `<font size="..." color="..." face="...">
11. `<h1>
12. `<h2>
13. `<h3>
14. `<h4>
15. `<h5>
16. `<h6>
17. `<i>
18. `<img src="...">
19. `<p>
20. `<small>
21. `<strike>
22. `<strong>
23. `<sub>
24. `<sup>
25. `<tt>
26. `<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.

---

940
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 — StackOverflow, 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.

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
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):
Rich Text

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:
In `onCreate()` of our activity, we find the EditText widget and designate the activity itself as being an `OnEditorActionListener` for the EditText:

```java
@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):

```java
@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);
```
The `searchFor()` method is where the formatting is applied to our search text:

```java
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`
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 292: The RichTextSearch sample, after searching on “can”](image)

**Saving Rich Text**

`SpannableString` and `SpannedString` are not `Serializable`. There is no built-in way to persist them directly.
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.
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.
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:

```xml
<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 an upcoming 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
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:

```xml
<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
<?xml version="1.0" encoding="utf-8"?>
<!-- */
/** //device/apps/common/res/drawable/status_icon_background.xml
**
** Copyright 2008, The Android Open Source Project
**
```
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
CUSTOM DRAWABLES

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
<?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
---
```
CUSTOM DRAWABLES

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_disable" />
  <item 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
CUSTOM DRAWABLES

- 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.

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
<?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">
    <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.

**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
<?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.-->```
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 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
<level-list>
  <!-- device/apps/common/res/drawable/stat_sys_battery.xml -->
  </level-list>
</transition>
```

**See the License for the specific language governing permissions and limitations under the License.**
CUSTOM DRAWABLES

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 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.
CUSTOM DRAWABLES

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
<?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%"/>
```

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.

```xml
<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., `setImageViewLevel()`) 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.

Seeing It In Action

To see these effects, take a look at the [Drawable/ScaleClip sample project](https://commonsware.com). This is derived from [an earlier example](https://commonsware.com) 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:

```xml
<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"/>

</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:

```java
package com.commonsware.android.scaleclip;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.SeekBar;
import com.actionbarsherlock.app.SherlockFragment;

abstract public class FragmentBase extends SherlockFragment implements SeekBar.OnSeekBarChangeListener {

    abstract void setBackgroundImage(ImageView image);

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {

        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);
        setBackgroundImage(image);
        image.setImageLevel(bar.getProgress());

    }

}
```
CUSTOM DRAWABLES

```java
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:

```java
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-xxdpi/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:

![ScaleDrawable, Level of 10000](image)

*Figure 293: ScaleDrawable, Level of 10000*

As we start sliding the SeekBar thumb to the left, the image shrinks progressively:
Figure 294: ScaleDrawable, Level of Approximately 5000

It eventually tends towards the 50% level specified in our android:scaleHeight and android:scaleWidth values:
Sliding it all the way to the left, though, causes the image to vanish.

The ClipDrawable starts off looking much like the ScaleDrawable:
As we slide the SeekBar to the left, the right side of the image gets clipped:
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
<?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.
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, in an ActionBarSherlock-equipped project. 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:

```java
package com.commonsware.android.shape;

import android.os.Bundle;
import android.support.v4.app.FragmentTransaction;
import android.widget.ImageView;
import com.actionbarsherlock.app.ActionBar;
import com.actionbarsherlock.app.ActionBar.Tab;
import com.actionbarsherlock.app.ActionBar.TabListener;
import com.actionbarsherlock.app.SherlockActivity;

public class MainActivity extends SherlockActivity 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=getSupportActionBar();
  }
}
```
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) {
    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>
CUSTOM DRAWABLES

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.

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
<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:
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:
That gives us:

![ShapeDrawable, Gradient Fill Rectangle](image)

Figure 299: 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.
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
<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:
Figure 300: 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
<?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
```
This gives us the following:

![Figure 301: ShapeDrawable, Gradient Fill Rounded Rectangle with Dashed Border](image)

### <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.
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.
CUSTOM DRAWABLES

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
<?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.
This gives us:

![Figure 302: ShapeDrawable, Ring with Gradient Fill](image)

**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](https://commonsware.com) sample project.

Our activity’s layout is just a `LinearLayout`, set to fill the screen:

```xml
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/widget"
```
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:

```java
package com.commonsware.android.tilemode;

import android.os.Bundle;
import android.support.v4.app.FragmentTransaction;
import android.view.View;
import com.actionbarsherlock.app.ActionBar;
import com.actionbarsherlock.app.ActionBar.Tab;
import com.actionbarsherlock.app.ActionBar.TabListener;
import com.actionbarsherlock.app.SherlockActivity;

public class MainActivity extends SherlockActivity 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 = getSupportActionBar();
        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
    }
}
```

CUSTOM DRAWABLES
CUSTOM DRAWABLES

```java
@Override
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`:

```xml
<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:

![BitmapDrawable, Without android:tileMode](image)

*Figure 303: BitmapDrawable, Without android:tileMode*

The `res/drawable/clamp.xml` resource, used on the second tab, adds `android:tileMode="clamp"`:

```xml
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
    android:src="@drawable/hatch"
    android:tileMode="clamp"/>
```
This causes the right-most column of pixels and the bottom-most column of pixels to be repeated to fill the available space:

![BitmapDrawable, Clamped](image)

*Figure 304: BitmapDrawable, Clamped*

Zooming in on the upper-left portion of our `LinearLayout` demonstrates this:
The res/drawable/repeat.xml resource, used on the third tab, employs android:tileMode="repeat":

```xml
<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:
Zooming in on an arbitrary chunk of the `LinearLayout` shows this effect:

*Figure 307: Portion of `BitmapDrawable`, Repeated*
The res/drawable/mirror.xml resource, used on the fourth tab, uses android:tileMode="mirror":

```xml
<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 308: BitmapDrawable, Mirrored](image)

Zooming in on an arbitrary chunk of the LinearLayout shows this effect:
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
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
  <item>
    <shape android:shape="rectangle">
      <gradient
```
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:
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
<?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">
```
We will get into how this works with a ProgressBar in a separate chapter.
XML Drawables and Eclipse

Alas, Eclipse has no special support for these drawables. When you double-click on one in the Package Explorer, you will get a standard XML editor, nothing more, at least at the present time.

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.

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.
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.

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 312: The stretch zones, as shown by a set of control lines to the left and top of the stretchable image](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.
CUSTOM DRAWABLES

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.

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 313: The draw9patch tool](image)

Eclipse, at the present time, does not have a built-in version of draw9patch, so Eclipse 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.
CUSTOM DRAWABLES

itself. For example, the image shown above, from the Drawable/NinePatch project, is 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
<?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>
```
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:

```java
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);
    }
}
```
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:
Figure 314: The NinePatch sample project, in its initial state

Figure 315: The NinePatch sample project, after making it bigger horizontally
Figure 316: The NinePatch sample application, after making it bigger in both dimensions
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.
For a widget whose name is `v`, on API Level 11 or higher, that is as simple as:

```java
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 we 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:
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 move 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:

```xml
<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>
```
In `onCreate()`, we load up the layout and get our hands on the `fadee` widget:

```java
@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:

```java
@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()`:

```java
@Override
public void onResume() {
    super.onResume();
    run();
}

@Override
public void onPause() {
```

---

**ANIMATORS**

1002
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, ViewPageIndicator, and other libraries, also wrote [NineOldAndroids](http://nineoldandroids.com). 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. You will need to download that project (look in the `library/` directory of the ZIP archive) and import it into Eclipse (if you are using Eclipse). The repository for this book has a compatible version of NineOldAndroids in its `external/` directory, and that version is what this chapter's samples will refer to.

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](http://nineoldandroids.com) sample app.
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:

```java
fadee.animate().alpha(0).setDuration(PERIOD);
```

we have:

```java
animate(fadee).alpha(0).setDuration(PERIOD);
```

This takes advantage of our static import:

```java
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`:

```java
ViewPropertyAnimator.animate(fadee).alpha(0).setDuration(PERIOD);
```

## The Foundation: Value and Object Animators

`ViewPropertyAnimator` itself is a layer atop 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
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:

```xml
<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:

```java
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"
    };
```

```
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.
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).
A flavor of `ofObject()` that takes the property name — akin to the `wordPosition` of `Int()` 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](https://commonsware.com), 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
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 317: Gmail Fragments (image courtesy of Google and AOSP)](image)

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
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 StackOverflow 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:

```xml
<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>
```
ThreePaneLayout itself is a subclass of LinearLayout, set up to always be horizontal, regardless of what might be set in the layout XML resource.

```java
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:

```java
@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:
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:
The `showLeft()` method simply performs those two animations in reverse:

```java
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:

```java
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) {  
```
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 on_create(), 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",
    };
"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;
        }
    }
}
```java
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:

![ThreePane, As Initially Launched](image)

*Figure 318: ThreePane, As Initially Launched*
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 319: ThreePane, After Clicking a Word](image)

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:
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
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.
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.
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.
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, the effect is akin to the Android menu system, with a container that slides up from the bottom and slides down and out when being removed. However, while menus are limited to menu choices, Android's animation framework lets one create a sliding panel containing whatever widgets you might want.

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.

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:

```java
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:

```java
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:
Figure 321: The SlidingPanel sample application, with the panel closed

... or open, in this case toggled via a menu choice in the SlidingPanelDemo activity:
**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”.

*Figure 322: The SlidingPanel sample application, with the panel open*
**Legacy Animations**

**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
<?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
loadAnimation() static method on the AnimationUtils class, seen here in our SlidingPanel constructor:

```java
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:
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
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:

```java
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.

Android 1.6 added some new interpolators. Notable are BounceInterpolator (which gives a bouncing effect as the animation nears the end) and OvershootInterpolator (which goes beyond the end of the animation range, then returns to the endpoint).

**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`:

```java
public void toggle() {
    TranslateAnimation anim = null;
    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()`).

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
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.
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.
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 90% 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.
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
```
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](https://cloud.google.com) 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](https://developer.android.com/guide/google/gcm/gcm.html))
- 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.
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 323: Android SDK Manager, Showing “Google Play services”](image-url)
Next, if you are an Eclipse user, 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.

Finally, when you create a project that is to use Maps V2, you will need to add a reference to that library project, either via the Eclipse project properties dialog or via the `android update lib-project` command. This is the same process that you use for adding other Android library projects, such as ActionBarSherlock.

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:

```java
-keep class * extends java.util.ListResourceBundle {
    protected Object[][] getContents();
}
```

It is unclear if this is strictly needed for Maps V2, as the Play Services library is used for other things beyond Maps V2.

**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
- Replace the reference to the ActionBarSherlock 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

This project uses Maps V2, and so it has a reference to that library project. It also uses ActionBarSherlock, so the action bar pattern can work on pre-Android 3.0 devices, so it has a reference to that library project as well:

![Figure 324: Android Library Projects Referenced by Maps V2/Basic]

The Manifest

Our manifest file is fairly traditional, though there are a number of elements in it that are required by Maps V2:

```xml
<?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="8"
    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
    android:glEsVersion="0x00020000"
    android:required="false"/>
</manifest>
```
<application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Sherlock.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="AIzaSyC4iyT46cB001dKGcy5EmAxK5uCQX20y8" />
  <meta-data
    android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version" />
  <activity android:name="LegalNoticesActivity" />
</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 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.
The Fragment and Activity

Our main activity — MainActivity — extends from the aforementioned AbstractMapActivity and simply overrides onCreate(), as most activities do:

```java
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:

```xml
<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.SupportMapFragment"/>
```

That resource, in turn, has a <fragment> element pointing to a com.google.android.gms.maps.SupportMapFragment class supplied by the Play Services library. This is a fragment that knows how to display a Maps V2 map. In particular, it is a fragment designed for use with the Android Support project’s backport of fragments. There is a corresponding com.google.android.gms.maps.MapFragment class for use with the native API Level 11 version of fragments.

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.

Note that SupportMapFragment is designed to work with the Android Support package, but not with ActionBarSherlock. However, a SherlockFragmentActivity (like AbstractMapActivity) can load a SupportMapFragment. So long as you are only using SupportMapFragment, it should co-exist with ActionBarSherlock without
difficulty. If, however, your objective would be to extend SupportMapFragment and add logic that might depend upon ActionBarSherlock (e.g., contribute items to the action bar), that will not work. Instead, you will need to use the Maps V2 version of the MapView widget directly (as will be covered later in this chapter) or hope that somebody else comes up with a SherlockMapFragment that works with Maps V2.

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 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:
If you choose the “Legal Notices” action bar item, the view shifts to show a bunch of 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, plus allow the user to toggle which set of map tiles to see.

The onCreate() method of the revised MapActivity is now a bit more involved:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {

```
super.onCreate(savedInstanceState);

if (readyToGo()) {
    setContentView(R.layout.activity_main);

    SupportMapFragment mapFrag = (SupportMapFragment) getSupportFragmentManager().findFragmentById(R.id.map);
    initListNav();
    map = mapFrag.getMap();

    if (savedInstanceState == null) {
        CameraUpdate center = CameraUpdateFactory.newLatLng(new LatLng(40.76793169992044, -73.98180484771729));
        CameraUpdate zoom = CameraUpdateFactory.zoomTo(15);

        map.moveCamera(center);
        map.animateCamera(zoom);
    }
}

After calling setContentView(), we can retrieve our SupportMapFragment via findFragmentById(), no different than any other fragment. In this case, we do so simply to call getMap() on it, which returns our GoogleMap object. 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:
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:

![Map V2 centered over Columbus Circle, New York City](image)

*Figure 327: Maps V2 Centered Over Columbus Circle, New York City*

We do all that work inside a check of the `savedInstanceState` parameter, as we only wish to initialize the map fragment when it is first created. Otherwise, on a configuration change, we re-center and re-zoom the map, wiping out any changes the user may have made. `MapFragment` (and `SupportMapFragment`) automatically retain their camera settings when a configuration change occurs, saving us that trouble.

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.

You will also notice that our action bar has list navigation, defaulted to “Normal”. That was set up by the initListNav() method invoked from onCreate():

```java
private void initListNav() {
    ArrayList<String> items = new ArrayList<String>();
    ArrayAdapter<String> nav = null;
    ActionBar bar = getSupportActionBar();

    for (int type : MAP_TYPE_NAMES) {
        items.add(getString(type));
    }

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH) {
        nav = new ArrayAdapter<String>(
            bar.getThemedContext(),
            android.R.layout.simple_spinner_item, items);
    } else {
        nav = new ArrayAdapter<String>(
            this,
            android.R.layout.simple_spinner_item, items);
    }

    nav.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
    bar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
    bar.setListNavigationCallbacks(nav, this);
}
```

Here, we simply set up a hard-coded set of navigation choices, from the MAP_TYPE_NAMES static array of string resource IDs:

```java
private static final int[] MAP_TYPE_NAMES = { R.string.normal,
                                              R.string.hybrid, R.string.satellite, R.string.terrain };
```

We also set up the activity itself as being the listener for navigation events, by passing it into setListNavigationCallbacks() and implementing the OnNavigationListener interface. This requires an onNavigationItemSelected() method, called when the user changes the value in the drop-down list:

```java
@Override
public boolean onNavigationItemSelected(int itemPosition, long itemId) {
    map.setMapType(MAP_TYPES[itemPosition]);
}
```
Here, based on the position selected in the list navigation, we call `setMapType()` on the `GoogleMap` with a corresponding map type:

```java
private static final int[] MAP_TYPES = {
    GoogleMap.MAP_TYPE_NORMAL,
    GoogleMap.MAP_TYPE_HYBRID,
    GoogleMap.MAP_TYPE_SATELLITE,
    GoogleMap.MAP_TYPE_TERRAIN
};
```

The `setMapType()` method switches the tile set used by the map. For example, we can switch to “hybrid” mode, which is a satellite view with labels for points of interest:

![Maps V2 with Hybrid Map Tiles](image)

*Figure 328: Maps V2 with Hybrid Map Tiles*

There is also a pure satellite mode (sans labels) and a “terrain” mode for a roughly topographical look at the landscape.

We also hold onto the navigation item across configuration changes via `onSaveInstanceState()` and `on RestoreInstanceState()`:
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 has four additional statements – calls to a private addMarker() method to define the four landmarks:

```java
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.

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.

If you look at the full implementation of onCreate(), you will see that our addMarker() calls are outside the savedInstanceState check that we added in MapsV2/NooYawk:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (readyToGo()) {
        setContentView(R.layout.activity_main);

        SupportMapFragment mapFrag = (SupportMapFragment) getSupportFragmentManager().findFragmentById(R.id.map);

        initListNav();

        map = mapFrag.getMap();

        if (savedInstanceState == null) {
            CameraUpdate center = CameraUpdateFactory.newLatLng(new LatLng(40.76793169992044, -73.98180484771729));

            CameraUpdate zoom = CameraUpdateFactory.zoomTo(15);

            map.moveCamera(center);
            map.animateCamera(zoom);
        }
    }
```
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:

![Maps V2 with Two Markers](image)

*Figure 329: 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:
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.
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:

```java
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:

```java
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 onCreate() moves the CameraUpdateFactory work until after all four of the addMarker() calls and changes it a bit:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (readyToGo()) {
        setContentView(R.layout.activity_main);

        SupportMapFragment mapFrag = (SupportMapFragment) getSupportFragmentManager().findFragmentById(R.id.map);
        initListNav();

        map = mapFrag.getMap();

        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);
    }
```
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:
The September 2013 update to the Play Services SDK exposed some new marker capabilities in this area: 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:
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:
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):

```java
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));
}
```
When we call `addMarker()`, we supply corresponding values:

```java
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 334: Maps V2 with Default Info Window](image)

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`
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
<?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:
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`.

Then, we just need to tell the `GoogleMap` to use our `InfoWindowAdapter`, via a call to `setInfoWindowAdapter()`, such as this statement from `onCreate()` of our new edition of `MainActivity`:
Now, when the user taps on a marker, they will get our customized info window:

![Figure 335: Maps V2 with Customized Info Window](image)

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:

```java
map.setOnInfoWindowClickListener(this);
```

This requires us to implement an `onInfoWindowClick()` method, where we are passed the `Marker` representing the tapped-upon info window:

```java
@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:
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 \texttt{View} you return for the info window is converted into a \texttt{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 in \texttt{later in this chapter}.

\section*{Images and Your Info Window}

The \texttt{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 \texttt{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.

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
<?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
<?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
            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"/>
    </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:

```java
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:

```java
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:

```java
addMarker(map, 40.748963847316034, -73.96807193756104, 
    R.string.un, R.string.united_nations, "UN_HQ.jpg");
addMarker(map, 40.76866299974387, -73.98268461227417, 
    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:

```java
public 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:

```java
@Override
public View getInfoContents(Marker marker) {
    View popup = contentsCache.get(marker.getId());

    if (popup == null) {
        popup = inflater.inflate(R.layout.popup, null);
        contentsCache.put(marker.getId(), popup);

        TextView tv = (TextView) popup.findViewById(R.id.title);
        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 {
            // Convert image to icon size
            // and set the image
```

---

**Mapping with Maps V2**

Subscribe to updates at https://commonsware.com

Special Creative Commons BY-NC-SA 4.0 License Edition
It attempts to retrieve an already-inflated layout for this info window from contentsCache, a HashMap maintained to hold onto already-created layouts:

```
private HashMap<String, Uri> images = null;
private Context ctxt = null;
```

Not only does this save some processing time by avoiding re-inflating a layout unnecessarily, but it also will help Picasso be able to populate the ImageView, as will be demonstrated shortly.

If the contentsCache has an inflated layout for this Marker, that View is returned, as it should already be set up (and possibly have an image, if there was one to be loaded and Picasso has completed its work).

If there was a cache miss, getInfoContents() starts by inflating the popup layout and populating the title and snippet as before. Then, getInfoContents() retrieves the icon ImageView from the inflated popup View, plus fetches 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:

```java
static class MarkerCallback implements Callback {
    Marker marker = null;

    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:

![Maps V2 with Popup and Thumbnail](image-url)

*Figure 337: Maps V2 with Popup and Thumbnail*
Note that the technique shown here will run into scaling issues with lots of markers... or, more accurately, with lots of info windows. Using a LruCache or some similarly-bounded cache for the layouts instead of a HashMap should help in this regard.

**Setting the Marker Icon**

Unlike Maps V1, 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:

```java
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.:
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
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:

```java
@Override
public boolean onMarkerClick(Marker marker) {
    Toast.makeText(this, marker.getTitle(), Toast.LENGTH_LONG).show();
    return(false);
}
```

![Figure 339: Maps V2 with Toast and Info Window](image)

Our call to setOnMarkerClickListener() is up in the onCreate() method of MainActivity:

```java
map.setOnMarkerClickListener(this);
```
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:

```java
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 onCreate():

```java
map.setOnMarkerDragListener(this);
```
That requires `MainActivity` to implement `OnMarkerDragListener`, which in turn requires three methods to be defined: `onMarkerDragStart()`, `onMarkerDrag()`, and `onMarkerDragEnd()`:

```java
@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
```

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.
Also, up in `onCreate()`, we retain our `SupportMapFragment` across configuration changes via `setRetainInstance(true)`: 

```java
mapFrag.setRetainInstance(true);
```

While `MapFragment` and `SupportMapFragment` will hold onto their markers across a configuration change by default, they do not hold onto changed marker positions via drag-and-drop. 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:

```java
package com.commonsware.android.mapsv2.model;
import android.content.Context;
```

---

1076
public class Model {
    String title;
    String snippet;
    double lat;
    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,
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):

```java
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);
    }
}
```
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:
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

```java
map.setMyLocationEnabled(true);
map.setOnMyLocationChangeListener(this);
```

The latter method is new to the February 2013 update to the Maps V2 portion of the Play Services Android library project. If, when trying to implement it, you get a compile error complaining that there is no such method, be sure to update your Play Services component in the SDK Manager and start using the updated Android library project.

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:

```java
@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:
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.

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 StackOverflow 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 four lines to the `onCreate()` implementation of `MainActivity` to configure the `GoogleMap`:

---

Figure 341: Maps V2, Showing the User's Location
locMgr=(LocationManager) getSystemService(LOCATION_SERVICE);
crit.setAccuracy(Criteria.ACCURACY_FINE);

map.setMyLocationEnabled(true);
map.getUiSettings().setMyLocationButtonEnabled(false);

The first two lines get access to a LocationManager and indicate that a Criteria object (initialized as a data member) should require fine accuracy. These come from the location tracking subsystem in Android.

The next line 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 onReshape() 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:

```java
@Override
public void onResume() {
  super.onResume();
  locMgr.requestLocationUpdates(0L, 0.0f, crit, this, 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).

The call to setLocationSource() 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:
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.

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:

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/Pol sample application, which is a clone of MapsV2/Popup with two additions:

- 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 onCreate() of MainActivity:

```java
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 addOptions() 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:

![Figure 342: Maps V2 with Polyline](image)

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:

![Figure 343: Maps V2 with Polyline and Polygon](image)

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.

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 onCreate() of MainActivity, we call setOnCameraChangeListener() on our GoogleMap, supplying MainActivity itself as the listener:

```java
map.setOnCameraChangeListener(this);
```
This requires MainActivity to implement OnCameraChangeListener and supply an implementation of onCameraChange():

```java
@Override
public void onCameraChange(CameraPosition position) {
    Log.d(getClass().getSimpleName(),
            String.format("lat: %f, lon: %f, zoom: %f, tilt: %f",
                          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 15:36:39.456: 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: 11.015625
12-26 15:36:45.726: D/MainActivity(31419): lat: 40.766083, lon: -73.982028, zoom: 15.875862, tilt: 38.671875
```
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 SupportMapFragment 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
<?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"
>
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`:

```java
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, getSupportFragmentManager()));
  }
}
```

`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.

```java
package com.commonsware.android.mapsv2.pager;

import android.content.Context;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
```
MapPageAdapter declares that there should be ten pages (in getCount()) and returns an instance of PageMapFragment for each page. PageMapFragment is a subclass of SupportMapFragment, and so is responsible for displaying our map:

```java
import android.support.v4.app.FragmentStatePagerAdapter;

public class MapPageAdapter extends FragmentStatePagerAdapter {
    Context ctxt = null;

    public MapPageAdapter(Context ctxt, FragmentManager mgr) {
        super(mgr);
        this.ctxt = ctxt;
    }

    @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);
    }
}
```

```java
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.SupportMapFragment;
import com.google.android.gms.maps.model.LatLng;
import com.google.android.gms.maps.model.MarkerOptions;

public class PageMapFragment extends SupportMapFragment {
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        GoogleMap map = getMap();

        if (savedInstanceState == null) {
            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);

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 SupportMapFragment 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 SupportMapFragment and
override onActivityCreated(), as getMap() should return a non-null value at this
point. 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
  protected boolean canScroll(View v, boolean checkV, int dx, int x,
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 SupportMapFragment 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:
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](https://commonsware.com) and related [bit of code in a GitHub Gist](https://commonsware.com). 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](https://commonsware.com).

A simple approach to computing the fractional `LatLng` would be to apply the fraction to the latitude and the longitude as Java double values:

```java
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
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.
Since we want to use the Google map utility library, we need to download and add it to our project by one means or another. The book’s GitHub repository has a copy of this library in the external/android-map-utils directory, though you are welcome to obtain yours by some other means, such as Maven. 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:

```java
private static final LatLng PENN_STATION = new LatLng(40.749972, -73.992319);
private static final LatLng LINCOLN_CENTER =
    new LatLng(40.7666299974387, -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:

```java
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:

```java
addMarker(map, 40.748963847316034, -73.96807193756104,
    R.string.un, R.string.united_nations);
addMarker(map, 40.76866299974387, -73.97989511489868,
    R.string.carnegie_hall, R.string.practice_x3);
```

1097
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):

```java
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):

```xml
<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():

```java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getSupportMenuInflater().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));
}
```

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:

```java
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:

```java
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 {
        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:

```java
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:
Figure 345: Maps V2 Animator Demo, As Initially Launched

Tapping that action bar item ("directions" icon) will reset the camera position and start animating the marker:
Two seconds later, the marker will reach its destination, presumably to board a train:
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

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.

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 (or, more accurately, SupportMapFragment). 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.

Also note that while MapView is a ViewGroup, you are not allowed to add child widgets to it.

**Maps and ActionBarSherlock**

All of the samples shown so far in this book have used ActionBarSherlock for their action bar. However, we have been using SupportMapFragment, which itself is not aware of ActionBarSherlock.

This actually works.

While the general pattern is to use a Sherlock version of a fragment class (e.g., SherlockListFragment), anything inheriting from the Android Support Library backport of Fragment will work with ActionBarSherlock... so long as it does not try to manipulate the action bar. SupportMapFragment, on its own, does not touch the action bar, and so we are safe.

However, what if we do want our maps to manipulate the action bar, such as adding tabs? Then, we need a SherlockFragment that also supports a GoogleMap. The Android Support Library is unlikely to supply such a class itself, and as of the time of this writing, neither does ActionBarSherlock.
However, it is possible to create such a class... something Alexandre Gherschon did in late 2012, with his take on a SherlockMapFragment implementation.

```java
package com.actionbarsherlock.app;

import android.app.Activity;
import android.support.v4.app.Watson.OnCreateOptionsMenuListener;
import android.support.v4.app.Watson.OnOptionsItemSelectedListener;
import android.support.v4.app.Watson.OnPrepareOptionsMenuListener;
import com.actionbarsherlock.app.SherlockFragmentActivity;
import com.actionbarsherlock.internal.view.menu.MenuItemWrapper;
import com.actionbarsherlock.internal.view.menu.MenuWrapper;
import com.actionbarsherlock.view.Menu;
import com.actionbarsherlock.view.MenuItem;
import com.google.android.gms.maps.SupportMapFragment;

public class SherlockMapFragment extends SupportMapFragment implements
        OnCreateOptionsMenuListener, OnPrepareOptionsMenuListener, OnOptionsItemSelectedListener {
    private SherlockFragmentActivity mActivity;

    public SherlockFragmentActivity getSherlockActivity() {
        return mActivity;
    }

    @Override
    public void onAttach(Activity activity) {
        if (!(activity instanceof SherlockFragmentActivity)) {
            throw new IllegalStateException(getClass().getSimpleName()
                    + " must be attached to a SherlockFragmentActivity.");
        }
        mActivity = (SherlockFragmentActivity) activity;

        super.onAttach(activity);
    }

    @Override
    public void onDetach() {
        mActivity = null;
        super.onDetach();
    }

    @Override
    public final void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
        onCreateOptionsMenu(new MenuWrapper(menu),
                mActivity.getSupportMenuInflater());
    }

    @Override
    public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
        // Nothing to see here.
    }
```
You can then extend SherlockMapFragment, creating your own fragment that can, among other things, manipulate the action bar, as seen in the MapsV2/Sherlock sample application:

```java
package com.commonsware.android.mapsv2.sherlock;

import android.os.Bundle;
import android.util.Log;
import com.actionbarsherlock.app.SherlockMapFragment;

public class MyMapFragment extends SherlockMapFragment {
  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    if (getMap() != null) {
      Log.d(getClass().getSimpleName(), "Map ready for use!");
    }
  }
}
```

One thing to be careful of, though, is the timing of when you try to use the GoogleMap object. For example, in onCreateView(), the GoogleMap is not yet ready for use. onActivityCreated(), as shown in the above code, appears to be a safe alternative.
You can then use your map fragment class wherever you had been using SupportMapFragment, such as in a layout resource:

```xml
<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.commonsware.android.mapsv2.sherlock.MyMapFragment"/>
```

### 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:

```java
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(getSupportFragmentManager(),
                TAG_ERROR_DIALOG_FRAGMENT);
    } else {
        Toast.makeText(this, R.string.no_maps, Toast.LENGTH_LONG).show();
        finish();
    }
}
```
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:

```java
// following from

/*
 * 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
            // GLES 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
                }
            }
        }
    }
    return false;
}
```
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:

```java
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);
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)
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:

```java
package com.commonsware.android.mapsv2.basic;

import android.os.Bundle;
import android.widget.TextView;
import com.actionbarsherlock.app.SherlockActivity;
import com.google.android.gms.common.GooglePlayServicesUtil;

public class LegalNoticesActivity extends SherlockActivity {
    @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:
Figure 348: Map with Many Markers (from Android Maps Extensions demo app)
Figure 349: Same Map with 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.
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.

Some developers report seeing black regions being left behind during animations of a map, such as sliding between ViewPager pages. Various workarounds are discussed in the linked-to StackOverflow question. At this time, the leading workaround, supplied by Jeff Gilfelt, involves putting a transparent frame on top of the map. According to Google engineers, the issue ties back to the GLSurfaceView used for drawing the map.

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:
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.
The official support point for Maps V2 for Android is StackOverflow. Questions tagged with *both android and google-maps* should show up on Google's radar.
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.

This chapter starts with a discussion of the various ways you can go about creating custom View classes. It then moves into an examination of ColorMixer, a composite widget, made up of several other widgets within a layout.

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:

![Image of toggle buttons](image)

*Figure 352: 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 `ColorWheel1` 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:

![Color Wheel](image)

*Figure 353: 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:
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="40dp"
       android:layout_height="40dp"
       android:layout_alignParentLeft="true"
       android:layout_centerVertical="true"
       android:layout_marginLeft="4dp"
  />
</merge>
```
CRAFTING YOUR OWN VIEWS

<SeekBar android:id="@+id/red"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/redLabel"
    android:layout_toRightOf="@id/swatch"
    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/swatch"
    android:layout_marginLeft="4dip"
    android:layout_marginTop="4dip"
    android:text="@string/green"
    android:textSize="10pt" />
<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/swatch"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="8dip" />

1123
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`:

```xml
<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`
Crafting Your Own Views

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:

```java
public ColorMixer(Context context) {
    this(context, null);
}

public ColorMixer(Context context, AttributeSet attrs) {
```
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:

```java
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:

```java
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();
    }
```
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).
So, for example, here are implementations of `onSaveInstanceState()` and `onRestoreInstanceState()` from ColorMixer:

```java
@Override
public Parcelable onSaveInstanceState() {
    Bundle state = new Bundle();
    state.putParcelable(SUPERSTATE, super onSaveInstanceState());
    state.putInt(COLOR, getColor());
    return state;
}

@Override
public void onRestoreInstanceState(Parcel 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`:

```java
public interface OnColorChangedListener {
    public void onColorChange(int argb);
}
```

ColorMixer also provides getters and setters for an `OnColorChangedListener` object:

```java
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:

```java
private SeekBar.OnSeekBarChangeListener onMix = new SeekBar.OnSeekBarChangeListener() {
    public void onProgressChanged(SeekBar seekBar, int progress, boolean fromUser) {
        int color = getColor();
    }
```
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
<?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.
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:

```java
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.
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:

```java
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);
            }
        } else {
            reset();
        }
    }
}
```

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
CRAFTING YOUR OWN VIEWS

- 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.:

```xml
<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:

```java
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() {
```
The result is much as you would expect: a countdown of the time remaining:

![ReverseChronometer](image)

*Figure 355: ReverseChronometer, Early in Countdown*

...changing to orange when we are within the warning duration:
...and changing to red when time has run out:
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.
**AspectLockedFrameLayout** 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](https://github.com/commonsguy/cwac-layouts). 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:

```java
public void setAspectRatioSource(View 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:

```java
// from com.android.camera.PreviewFrameLayout, with slight
// modifications
```
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:

```xml
<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>
```

**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](https://commonsware.com). 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:

- 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

**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.

```java
package com.commonsware.cwac.layouts;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.util.AttributeSet;

//import android.graphics.Rect;

public class MirroringFrameLayout extends AspectLockedFrameLayout {
    private Mirror mirror=null;
    private Bitmap bmp=null;
    private Canvas bmpBackedCanvas=null;

    public MirroringFrameLayout(Context context) {
        this(context, null);
    }

    public MirroringFrameLayout(Context context, AttributeSet attrs) {
        super(context, attrs);

        setWillNotDraw(false);
    }

    public void setMirror(Mirror mirror) {
        this.mirror=mirror;

        if (mirror != null) {
            mirror.setSource(this);
            setAspectRatioSource(mirror);

            // following needed in case mirror has not been sized
            // yet, so we can determine our aspect ratio

            post(new Runnable() {
                public void run() {
                    requestLayout();
                }
            });
        }
    }

    @Override
    public void draw(Canvas canvas) {
        if (mirror != null) {
```
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 Mirror that is connected to this MirroringFrameLayout. In addition to holding onto the Mirror in a mirror data member, we:

- Call setSource() on the Mirror, to tell it its associated MirroringFrameLayout
- Call setAspectRatioSource(), inherited from AspectLockedFrameLayout, so our contents will match the aspect ratio from that source
Use post() to arrange to call requestLayout() on ourselves later on, in case the Mirror has not been sized yet (meaning we do not know our aspect ratio — requestLayout() will trigger a fresh onMeasure() call on our AspectLockedFrameLayout superclass to fix our aspect ratio and draw our contents accordingly)

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, 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()
- If we have our Mirror, call invalidate() on it, to get it to redraw itself

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.
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 a setSource() method, used to connect the MirroringFrameLayout from which the Mirror can obtain what it is supposed to display.

The bulk of the “business logic” lies in onDraw(), plus a helper calcCenter() static method:

```java
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    if (source != null) {
        getDrawingRect(rect);

        Bitmap cache = source.getLastBitmap();

        if (cache != null) {
            calcCenter(rect.width(), rect.height(), cache.getWidth(),
                        cache.getHeight(), rect);
            canvas.drawBitmap(cache, null, rect, null);
        }
    }
}

// based upon http://stackoverflow.com/a/14679729/115145
private 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);
}
```

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()
CRAFTING YOUR OWN VIEWS

• 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

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:

```xml
<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"
      android:background="#8800FF00">
    <com.commonsware.cwac.layouts.ForcedMirroringFrameLayout
        android:id="@+id/source"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="center"
        android:background="#88FF0000">
      <WebView
          android:id="@+id/webkit"
          android:layout_width="match_parent"
          android:layout_height="match_parent"/>
    </com.commonsware.cwac.layouts.ForcedMirroringFrameLayout>
  </FrameLayout>
</LinearLayout>
```
In this case, we set the background of the FrameLayout holding our MirrorFrameLayout to green, to show how the MirrorFrameLayout 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 MirrorFrameLayout to enable the mirroring logic:

```java
package com.commonsware.cwac.layouts.demo;

import android.annotation.SuppressLint;
import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebView;
import com.commonsware.cwac.layouts.Mirror;
import com.commonsware.cwac.layouts.MirroringFrameLayout;

public class SimpleMirrorActivity extends Activity {
    @SuppressLint("SetJavaScriptEnabled")
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.simple_mirror);

        MirroringFrameLayout source = (MirroringFrameLayout)findViewById(R.id.source);
        Mirror target = (Mirror)findViewById(R.id.target);

        source.setMirror(target);

        WebView wv = (WebView)findViewById(R.id.webkit);

        wv.getSettings().setJavaScriptEnabled(true);
        wv.loadUrl("http://commonsware.com");
    }
}
```
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:

---

**Figure 358: MirroringFrameLayout Above Its Mirror**
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.
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.
CUSTOM DIALOGS AND PREFERENCES

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.

![ColorMixerDialog](image)

*Figure 360: The ColorMixerDialog*
Basic AlertDialog Setup

The ColorMixerDialog class is not especially long, since all of the actual color mixing is handled by the ColorMixer widget:

```java
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);

        setContentView(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);
    }
}
```
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:

```java
@Override
public void onClick(DialogInterface dialog, int which) {
    if (initialColor!=mixer.getColor()) {
        onSet.onColorChange(mixer.getColor());
    }
}
```
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:

```java
@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:

```java
@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
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:

```java
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:

```java
@override
protected View onCreateDialogView() {
    mixer=new ColorMixer(getContext());
    return mixer;
}
```
Then, in onBindDialogView(), we set the mixer's color to be lastColor, a private data member:

```java
@NonNull
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 0xFFFFA4C639 (a.k.a., Android green):

```java
@Override
protected Object onGetDefaultValue(TypedArray a, int index) {
    return a.getInt(index, 0xFFFFA4C639);
}
```
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:

```java
@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:

```java
@Override
protected void onDialogClosed(boolean positiveResult) {
    super.onDialogClosed(positiveResult);

    if (positiveResult) {
        if (callChangeListener(mixer.getColor())) {
            lastColor = mixer.getColor();
            persistInt(lastColor);
        }
    }
}
```
CUSTOM DIALOGS AND PREFERENCES

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:

```xml
<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- Auuuuuuugh!" />
</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:
When tapped, it brings up the mixer:

*Figure 361: A PreferenceActivity, showing the ColorPreference*
Choosing a color and clicking “Set” persists the color value as a preference.
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).
PROGRESS INDICATORS
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 363: 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:

1162

Subscribe to updates at https://commonsware.com

Special Creative Commons BY-NC-SA 4.0 License Edition


PROGRESS INDICATORS

Figure 364: 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 365: 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 366: Android 4.0 ProgressBar, Horizontal Style, Indeterminate and Specific

Figure 367: Android 2.3.3 ProgressBar, Horizontal Style, Indeterminate and Specific
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 368: Android 4.0 ProgressBar, Horizontal Style, Primary-Only and Primary-Plus-Secondary](image)

![Figure 369: Android 2.3.3 ProgressBar, Horizontal Style, Primary-Only and Primary-Plus-Secondary](image)

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:
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:

```java
package com.commonsware.android.progress;
import android.app.Activity;
```
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);
        }
    }
}
PROGRESS INDICATORS
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.

1167

Subscribe to updates at https://commonsware.com

Special Creative Commons BY-NC-SA 4.0 License Edition


PROGRESS INDICATORS

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):

![Android Holo Colors Generator, Basic Info](image)

You can then toggle on and off which widgets you intend to use, so the generator will create custom styles for them:
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:

```xml
<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">
```
That theme, defined in apptheme_themes.xml, points to style resources for horizontal ProgressBar widgets:

```xml
<?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
<?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
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
<?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.
PROGRESS INDICATORS

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:

![Custom Progress Bar Style](image)

*Figure 372: 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
<?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"
        />
      </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
<?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" />
</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
<?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:
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
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 `ProgressDialog` 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:

```java
@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:
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.

```java
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);
  }
}
```
**PROGRESS INDICATORS**

Up front, as the first thing that you do in your `onCreate()` call, you need to call:

```java
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 ActionBarSherlock. Hence, when you run it on an older Android environment, you get a classic title bar with the progress indicator on the right:

![Progress Indicator in Title Bar](image)

*Figure 375: 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:
Now, you will notice a slightly unusual structure to the `onCreate()` method in the above code listing. Normally, the call to `super.onCreate()` is the very first thing that you want to do. And, ordinarily, you would only need to call `requestFeature()` on your window before the call to `setContentView()`. However, Jake Wharton has indicated that the code sequence shown above is the correct one when using ActionBarSherlock and its progress indicator.

**Action Bar Refresh-and-Progress Items**

One common pattern in an Android application is to have some sort of “refresh” action bar item, that causes you to do some work once it is tapped. For example, the Gmail app has a refresh action bar item that goes and checks for new email.

A handy way to visually represent that work is to replace the static action bar item icon with some sort of circular progress indicator while that work is going on. This can not only tell the user that we are working on their request, but it can also convey to the user that tapping it again is unlikely to be especially useful.

To see how this works, take a look at the `Progress/ActionBar` sample project.
**Progress Indicators**

This is a trimmed-down version of the book’s original action bar sample. It retains the “refresh” action bar item but gets rid of the others. The “refresh” action bar item will not actually refresh anything, but it will pretend to do work for a few seconds, replacing itself with a small indeterminate ProgressBar while that is going on.

In onCreateOptionsMenu() of our activity, we do the normal inflate-the-menu-resource work, plus hold onto the MenuItem associated with our refresh option:

```java
@override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);

    refresh = menu.findItem(R.id.refresh);

    return (super.onCreateOptionsMenu(menu));
}
```

In onOptionsItemSelected(), we call a private refresh() method if the user taps on our refresh action bar item:

```java
@override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.refresh) {
        refresh();

        return (true);
    }

    return (super.onOptionsItemSelected(item));
}
```

That refresh() method is where we do the work to change our action bar item to a progress indicator and pretend to do the fake work:

```java
private void refresh() {
    refresh.setActionView(R.layout.refresh);

    getListView().postDelayed(new Runnable() {
        public void run() {
            refresh.setActionView(null);
        }
    }, 5000);
}
```

We call setActionView() to replace the default action bar toolbar button with a custom View, inflated from res/layout/refresh.xml:
This is a `LinearLayout`, styled to match an action button, with a small `ProgressBar` widget in the center.

We then delay for 5000 millisecond (pretending to do work) before calling `setActionView(null)` to remove our inflated layout and return to the normal action bar button.

So while the activity starts with a conventional button...

![Figure 377: Normal Action Bar Button](image)
...once we tap it, the button is replaced with our progress indicator:

![Image of progress indicator replacing action bar button](image)

*Figure 378: Progress Indicator Replacing Action Bar Button*

The on-click listener that the action bar puts on the button remains on the button — we do not get control when the user taps on the `ProgressBar`, nor does the user get any visual feedback suggesting that such taps have meaning. Hence, while the refresh is going on, the user cannot request another refresh operation. If, for some reason, you actually want that behavior, you could set that up yourself with your inflated layout and event listeners on the widgets.

This sample implementation is not complete — for example, we should track whether our work is going on and toggle the action bar item to the progress indicator on a configuration change. But the basics are there and, as you can see, are fairly simple.

### 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.
PROGRESS INDICATORS

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.
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.

Custom Views: or How Those Progress Bars Work

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 Progress.Bar 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 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 Progress.Bar 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.
ADVANCED NOTIFICATIONS

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 SillyService, that will simulate doing real work in the background and maintaining a Notification along the way:

```java
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 obliged:

```java
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.

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:

```java
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 notification = 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.

**The ProgressBar**

Our `buildContent()` method just returns a `RemoteViews` object:

```java
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`:

```xml
<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:

```java
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;
}
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`.

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`:

```java
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:

```xml
<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:

```java
private PendingIntent buildContentIntent() {
    Intent i = new Intent(Settings.ACTION_SETTINGS);
    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:

```java
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 379: The custom ticker in our Notification, as seen on a Honeycomb tablet](https://commonsware.com/AndroidNotifications/fig379.png)

Eventually, the ticker vanishes, leaving us with the traditional system bar icon:

![Figure 380: The system bar icon for our Notification, as seen on a Honeycomb tablet](https://commonsware.com/AndroidNotifications/fig380.png)
Tapping on the icon brings up the notification drawer, with our custom content, including our ProgressBar:

![Notification ProgressBar, on Android 3.x Tablet](image)

*Figure 381: 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:

![Notification ProgressBar, on Android 4.x Phone](image)

*Figure 382: 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()`:  

---

**ADVANCED NOTIFICATIONS**

---

Subscribe to updates at https://commonsware.com Special Creative Commons BY-NC-SA 4.0 License Edition
@Override
protected  void onHandleIntent(Intent intent) {
    NotificationManager mgr =
        (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    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
ADVANCED NOTIFICATIONS

- Remove the ProgressBar via `setProgress(0, 0, false)`
- Use `setOngoing(false)` to allow the user to get rid of the Notification

When we run this sample, we see our `ProgressBar`-based Notification at the outset:

![Figure 383: Notification with ProgressBar](image)

When the loop is finished, we see the Notification as updated for its final state:
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.
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.
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`. This chapter also assumes that you have read the chapter on action bar navigation.

ViewPager with Action Bar Tabs

More often than not, if you wish to use tabs in concert with your `ViewPager`, you will use `PagerTabStrip`, or perhaps an indicator from the `ViewPagerIndicator` project, to supply those tabs. Those are designed to integrate cleanly with `ViewPager` and were demonstrated earlier in this book.

And, as was outlined in the chapter on action bar navigation, while you can request to use tabs in your action bar, those tabs do not necessarily work well — for example, sometimes they will convert into a drop-down list instead.

That being said, perhaps there are scenarios in which you want to use action bar tabs to control pages in a `ViewPager`. This is certainly possible, though it requires teaching the action bar about the `ViewPager` and teaching the `ViewPager` about the action bar.
To see what is required, take a look at the [ViewPager/TabPager](#) sample project. This is based on the [ViewPager/Fragments](#) and [ViewPager/Indicator](#) sample projects reviewed in the chapter introducing ViewPager.

As with those projects, our activity hosts a ViewPager, which we intend to populate with EditorFragments by way of a SampleAdapter. None of that has changed in this new sample. However, we do more work in the activity (ViewPagerFragmentDemoActivity) that ties the pager into the action bar.

### Tying Tabs to Pages

When the user taps on a tab, we expect the ViewPager to jump to the associated page, where “the associated page” is based on tab position. For example, tapping the second tab should bring up the second page. To change the current page of a ViewPager, you simply need to call `setCurrentItem()` — we just need to know when to call that and what value to supply as the position.

To do that, as we set up the action bar tabs, we attach the index of the tab as the tab’s tag via `setTag()`:

```java
ActionBar bar = getSupportActionBar();
bar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

for (int i = 0; i < 10; i++) {
    bar.addTab(bar.newTab()
        .setText("Editor #" + String.valueOf(i + 1))
        .setTabListener(this).setTag(i));
}
```

We also set the activity itself as being the listener, meaning that we have to implement the `TabListener` interface on the activity. That requires three methods to be added: `onTabSelected()`, `onTabReselected()`, and `onTabUnselected()`:

```java
@Override
public void onTabSelected(Tab tab, FragmentTransaction ft) {
    Integer position = (Integer)tab.getTag();
    pager.setCurrentItem(position);
}

@Override
public void onTabUnselected(Tab tab, FragmentTransaction ft) {
    // no-op
}
```

---

1198
public void onTabReselected(Tab tab, FragmentTransaction ft) {
    // no-op
}

We ignore onTabReselected() and onTabUnselected(), providing stub implementations because the interface requires them. However, we have a real implementation of onTabSelected(), one which grabs the tab’s position out of its tag and uses that in the call to setCurrentItem() on the ViewPager. As a result, if the user chooses a tab — including choosing something from the drop-down that replaces tabs in some circumstances — the pager updates to match.

**Tying Pages to Tabs**

Obviously, we needed to have tapping on a tab bring up its associated page. What might be less obvious at the outset is that the reverse is true: we need to select the right tab if the user navigates to another page via swiping in the ViewPager. Otherwise, horizontal swipe actions will show a different page than the currently-selected tab would indicate.

Fortunately, this too is relatively easy.

As part of our ViewPager setup, we have it send page-scrolled events to our activity via setOnPageChangeListener():

```java
pager=ViewPager.findViewByld(R.id.pager);
pager.setAdapter(new SampleAdapter(getSupportFragmentManager()));
pager.setOnPageChangeListener(this);
```

That requires our activity to implement the ViewPager.OnPageChangedListener interface, which in turn requires three methods:

1. onPageScrolled()
2. onPageScrollStateChanged()
3. onPageSelected()

It is the latter one that we care about — this will be called when the user fully swipes to another page. In our implementation, we simply set the selected tab to the same position:

```java
@Override
public void onPageSelectedStateChanged(int arg0) {
    // no-op
}
```
MORE FUN WITH PAGERS

```java
@override
public void onPageScrolled(int arg0, float arg1, int arg2) {
    // no-op
}

@override
public void onPageSelected(int position) {
    getSupportActionBar().setSelectedNavigationItem(position);
}
```

The Results

The visual effect, normally, is not that different from using a PagerTabStrip:

![Figure 385: ViewPager with Action Bar Tabs, Portrait/Normal Screen](image)

The user can swipe the main area to move between pages, with the tabs automatically updating. The user can also tap on a tab to move to that page, or swipe the tabs to reach a tab not presently visible.

However, since these are action bar tabs, not a PagerTabStrip, they will automatically convert into a list-navigation-style Spinner, at various times:
Also, when tabs are “collapsed” into the list navigation mode, a bug in Android means that swiping the pager does not update the Spinner, as setSelectedNavigationItem() no longer works there.

Using ViewPagerIndicator

In addition to using action bar tabs, 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), another project from Jake Wharton (author of ActionBarSherlock).

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, like ActionBarSherlock, is open source, released under the Apache Software License 2.0.
**Downloading VPI**

ViewPagerIndicator is distributed in much the same fashion as is ActionBarSherlock. On the [ViewPagerIndicator home page](https://commonsware.com) you will find buttons to download a ZIP or a TGZ containing the library and sample code.

The library, as with ActionBarSherlock, is an Android library project. 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 the import wizard.

And, as with any other Android library project, you will need to attach the library to your main project. If you are using Eclipse, this will be via Project > Properties > Android; from the command-line, you would use the `android update project` command.

**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](https://commonsware.com) 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
<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>
```
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:

```java
package com.commonsware.android.pager3;
import android.os.Bundle;
import android.support.v4.view.PagerAdapter;
import android.support.v4.view.ViewPager;
import com.actionbarsherlock.app.SherlockFragmentActivity;
import com.viewpagerindicator.TabPageIndicator;

public class MainActivity extends SherlockFragmentActivity {

    @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, getSupportFragmentManager());
    }
}
```

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.
MORE FUN WITH PAGERS
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., @style/Theme.Sherlock.Light), 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"/>
/>
<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="@style/Theme.Sherlock.Light">
>
<item name="vpiTabPageIndicatorStyle">
>@style/TabStyle</item>
</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>
<item name="android:textSize">
>14sp</item>
</item>
<item name="android:textStyle">
>italic</item>
</item>
<item name="android:paddingLeft">
>16dp</item>
</item>
<item name="android:paddingRight">
>16dp</item>
</item>
<item name="android:fadingEdge">
>horizontal</item>
</item>
<item name="android:fadingEdgeLength">
>8dp</item>
</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.

1204

Subscribe to updates at https://commonsware.com

Special Creative Commons BY-NC-SA 4.0 License Edition


MORE FUN WITH PAGERS

With the custom style as defined above — specifying green italicized tab titles – we get tabs that look like these:

![View Pager with Customized TabPagerIndicator](image)

*Figure 387: 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.

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
<?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"/>
</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
<?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">
```

1206
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:
@Override
public int getCount() {
    return 3;
}

@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:

```java
package com.commonsware.android.pagercolumns;

import android.os.Bundle;
import android.support.v4.app.FragmentPagerAdapter;
import android.support.v4.view.ViewPager;
import com.actionbarsherlock.app.SherlockFragmentActivity;
import com.viewpagerindicator.TabPageIndicator;

public class MainActivity extends SherlockFragmentActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        ViewPager pager = (ViewPager) findViewById(R.id.pager);

        if (pager == null) {
            if (getSupportFragmentManager().findFragmentById(R.id.editor1) == null) {
                FragmentPagerAdapter adapter = buildAdapter();

                getSupportFragmentManager().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);
}

private FragmentPagerAdapter buildAdapter() {
    return new SampleAdapter(this, getSupportFragmentManager());
}

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 388: ViewPager with Customized TabPagerIndicator. Again.](image)

However, on a tablet, we get our three editors side-by-side:
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.
Introduction to 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

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, which you can download and add to your project by conventional means (e.g., putting it in the libs/ directory).

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.

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:

```java
static class SamplePagerAdapter extends ArrayPagerAdapter {
  public SamplePagerAdapter(FragmentManager fragmentManager,
                              ArrayList<PageDescriptor> descriptors) {
    super(fragmentManager, descriptors);
  }

  @Override
  protected EditorFragment createFragment(PageDescriptor desc) {
    return EditorFragment.newInstance(desc.getTitle());
  }
}
```

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:

```java
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.

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.

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 ActionBarSherlock, 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.
MORE FUN WITH PAGERS

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.

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:

```java
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 (createFragment.newInstance(title));
    }
}
```

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:

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ViewPager pager=(ViewPager)findViewById(R.id.pager);

    if (pager == null) {
        if (getSupportFragmentManager().findFragmentById(R.id.editor1) == null) {
```
The `buildAdapter()` method changes a bit, to create our `ArrayPagerAdapter` subclass using an array of `SimplePageDescriptor` objects:

```java
private SamplePagerAdapter buildAdapter() {
    ArrayList<PageDescriptor> pages = new ArrayList<PageDescriptor>();
    for (int i=0; i<3; i++) {
        pages.add(new SimplePageDescriptor(buildTag(i), buildTitle(i)));
    }
    return new SamplePagerAdapter(getSupportFragmentManager(), pages);
}
```

`buildAdapter()`, in turn, uses `buildTag()` and `buildTitle()` methods to retrieve the tag and title to use given a position:

```java
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:

```java
private void populateColumn(FragmentManager fm,
                           FragmentTransaction ft,
```
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 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:
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”):
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:

```java
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);
        }
    }
}
```
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:

```java
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:

```java
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
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:

```java
package com.commonsware.cwac.pager;

import android.os.Parcelable;

public interface PageDescriptor extends Parcelable {
  String getFragmentTag();

  String getTitle();
}
```
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:

```java
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);
    }
}
```
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:

```java
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 Parcelable 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.
RetentionStrategy

ArrayPagerAdapter also uses aRetentionStrategy, 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:

```java
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 ofFragmentPagerAdapter, keeping all fragments around once created, and merely attach()-ing and detach()-ing them from the FragmentManager as dictated:

```java
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) {
        currTransaction.detach(fragment);
    }
};
```

A future implementation ofArrayPagerAdapter should include another strategy that behaves more likeFragmentStatePagerAdapter, removing the fragments entirely and allowing them to be garbage collected, while usingPageEntry 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 usesJava generics to allow developers to state what Fragment subclass the pages are. This is for use with convenience methods — getExistingFragment() andgetCurrentFragment() — to help reduce the developer's need to downcast those Fragment instances to some subclass. If the pages in theViewPager will all come from a single Fragment subclass, the developer
would use that class as the T in the declaration; otherwise, the developer would just use Fragment:

\[
\text{abstract public class } \text{ArrayPagerAdapter<T extends Fragment> extends PagerAdapter {}}
\]

### 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):

\[
\text{public ArrayPagerAdapter(FragmentManager fragmentManager, ArrayList<PageDescriptor> descriptors, RetentionStrategy retentionStrategy) }
\]
\[
\text{this.fm=fragmentManager;}
\]
\[
\text{this.entries=new ArrayList<PageEntry>();}
\]
\[
\text{for} \ (\text{PageDescriptor desc : descriptors)} \ {\ }
\]
\[
\text{validatePageDescriptor(desc)};
\]
\[
\text{entries.add(new PageEntry(desc));}
\]
\[
\text{this.retentionStrategy=retentionStrategy;}
\]
\[
\text{if} \ (\text{this.retentionStrategy == null)} \ {\ }
\]
\[
\text{this.retentionStrategy=KEEP;}
\]

### 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.
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:

```java
@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:

```java
@Override
public String getPageTitle(int position) {
    return entries.get(position).getDescriptor().getTitle();
}
```

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
MORE FUN WITH PAGERS

- 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”

```java
@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:

```java
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
@Override
public void destroyItem(ViewGroup container, int position, Object object) {
    if (currTransaction == null) {
        currTransaction = fm.beginTransaction();
    }
    currTransaction.remove(object);
    // More fun with Pagers...
}
```
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:

```java
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
@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):

```java
@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
MORE FUN WITH PAGERS

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:

```java
@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:

```java
@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).

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():

```java
@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 getItemPosition(). 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.

g.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)

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:

```java
@override
public int getItemPosition(Object o) {
    Integer result = positionDelta.get(o);
    if (result == null) {
        return PagerAdapter.POSITION_UNCHANGED;
    }
    return result;
}
```

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():

```java
public void add(PageDescriptor desc) {
    validatePageDescriptor(desc);
    positionDelta.clear();
    entries.add(new PageEntry(desc));
    notifyDataSetChanged();
}
```

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
public void insert(PageDescriptor desc, int position) {
    validatePageDescriptor(desc);

    positionDelta.clear();

    for (int i=position; i < entries.size(); i++) {
        Fragment f=getExistingFragment(i);

        if (f != null) {
            positionDelta.put(f, i + 1);
        }
    }

    entries.add(position, new PageEntry(desc));
    notifyDataSetChanged();
}

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:

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);
        }
    }

    entries.remove(position);
    notifyDataSetChanged();
}

Finally, a move() is simply treated as a remove() from the old position and an insert() of the same page into the new position:
public void move(int oldPosition, int newPosition) {
    if (oldPosition != newPosition) {
        PageDescriptor desc = entries.get(oldPosition).getDescriptor();
        remove(oldPosition);
        insert(desc, newPosition);
    }
}

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

public T getCurrentFragment() {
    return (currPrimaryItem);
}

Both are set to return the generic type T that the developer uses when creating a subclass of ArrayPagerAdapter.
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 Google 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.
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 Google TV or the OUYA console
- Devices that have dedicated keyboard accessories, such as the keyboard “slice” available for the ASUS Transformer 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
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.
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:

```xml
<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>
```
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 392: Focus Sampler, Showing Requested Focus](image)

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`. 
In this activity, our layout (focusable_button) is a bit different:

```xml
<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"/>

    <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:

```java
@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);
}
```
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:

![Focus Sampler, Showing Manually-Requested Focus](image)

We also wire up the Button to the activity for click events, and in onClick(), we call clearFocus() to abandon the focus:
What `clearFocus()` will do is return to the original default focus for this activity, in our case the first `EditText`:

![Figure 394: Focus Sampler, After Clearing the Focus](image)

**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`).
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
<?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.
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 395: A Split Action Bar](image)

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:
Override 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?)
• “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
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](https://developer.android.com/training/accessibility/accessibility-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
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](https://commonsware.com) 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:

```xml
<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:textStyle="bold"
        android:textSize="30px"
        tools:context=".MainActivity"/>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/normal_30dp"
        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: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:textStyle="bold"
        android:textSize="30sp"
        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:
As you can see, 30dp and 30sp are equivalent.

If we raise the font scale to “large”, the 30sp text grows to match:
Moving to “huge” scale increases the 30sp text size further:
In the other direction, some users may elect to drop their font size to “small”, with a corresponding impact on the 30sp text:
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.
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.
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.
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

1258
• 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)
While well-written GUI frameworks are better organized than [XKCD's take on home organization](https://xkcd.com/543/), 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:
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():

```java
requestWindowFeature(Window.FEATURE_NO_TITLE);
getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                      WindowManager.LayoutParams.FLAG_FULLSCREEN);
```

The first statement removes the title bar (and also removes the action bar if you are using ActionBarSherlock). 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:

```xml
<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"/>
    </RadioGroup>
</RelativeLayout>
```
...while on Android 4.1 or higher, we have another two possibilities:

```xml
<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"
  </RadioGroup>
</RelativeLayout>
```
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"/>

<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>
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:
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:
**MISCELLANEOUS UI TRICKS**

*Figure 403: Sample UI, Lights-Out Mode, Nexus 4/Android 4.2*

*Figure 404: Sample UI, Lights-Out Mode, Nexus 10/Android 4.2*
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:

![Sample UI, Hidden-Navigation Mode, Nexus 4/Android 4.2](image)

*Figure 405: Sample UI, Hidden-Navigation Mode, Nexus 4/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:
**Figure 407:** Sample UI, “Full-Screen” Mode, Nexus 4/Android 4.2

**Figure 408:** Sample UI, “Full-Screen” Mode, Nexus 10/Android 4.2
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 409: Sample UI, True Full-Screen Mode, Nexus 4/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.
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
accomplish this, we will use a `postDelayed()` loop, to get control every 15 seconds to see if there has been user activity:

```java
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
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.
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. Here, we will look at some event bus alternatives for Android, how they can help us manage communications within our app, and how we can employ some of them.

Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the chapters on 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.
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 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.

Standard Intents as Event Bus

You can think of the standard Intent and <intent-filter> system as a three-channel event bus:

- 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`.

**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() {
    super.onResume();

    IntentFilter filter=new IntentFilter(NoticeService.BROADCAST);
    LocalBroadcastManager.getInstance(this).registerReceiver(onNotice, filter);
}
```
LocalActivity unregisters for this broadcast in `onPause()`:

```java
@Override
public void onPause() {
    super.onPause();
    LocalBroadcastManager.getInstance(this).unregisterReceiver(onNotice);
}
```

The `BroadcastReceiver` simply updates a `TextView` with the current date and time:

```java
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](https://commonsware.com/Android/). 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 saw a version of that logic in [the chapter on Notifications](https://commonsware.com/Android/), but it used regular (albeit ordered) broadcasts.

In the [EventBus/LocalBroadcastManager](https://commonsware.com/Android/) sample project, we blend:

- 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:

```java
package com.commonsware.android.eventbus.lbm;
import android.os.Bundle;
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class EventDemoActivity extends SherlockFragmentActivity {
    @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 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:

```java
package com.commonsware.android.eventbus.lbm;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;
```

1282
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;
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().sendBroadcast(event)) {
        NotificationCompat.Builder b = new NotificationCompat.Builder(this);
        b.setAutoCancel(true).setDefaults(Notification.DEFAULT_SOUND)
            .setContentTitle(getString(R.stringnotif_title))
            .setContentText(Integer.toHexString(random))
            .setSmallIcon(android.R.drawable.stat_notify_more)
            .setTicker(getString(R.stringnotif_title))
            .setContentIntent(PendingIntent.getActivity(this, 0, ui, 0));

        NotificationManager mgr = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);

        mgr.notify(NOTIFY_ID, b.build());
    }
}
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:

```java
package com.commonsware.android.eventbus.lbm;

import android.annotation.SuppressLint;
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;
import com.actionbarsherlock.app.SherlockListFragment;

public class EventLogFragment extends SherlockListFragment {
    static final String EXTRA_RANDOM = "r";
    static final String EXTRA_TIME = "t";
    static final String ACTION_EVENT = "e";
    private EventLogAdapter adapter = null;

    @Override
    public void onActivityCreated(Bundle state) {
        super.onActivityCreated(state);
    }
}
```
setRetainInstance(true);
gListView().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);
    }
}

private BroadcastReceiver onEvent = new BroadcastReceiver() {
    @Override

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 411: LocalBroadcastManager as Event Bus, Demo Activity](image)

Whereas if events are processed while the activity is not in the foreground, a `Notification` will be shown with the last results:
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 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:

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.

With LocalBroadcastManager, subscribers are BroadcastReceiver, 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
Event Buses

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:

```java
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:

```java
@Override
protected void doWakefulWork(Intent intent) {
```

1291
Here, we are posting an instance of a `RandomEvent`:

```java
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`:

```java
@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()`:

```java
@Subscribe
public void onRandomEvent(final RandomEvent event) {
    if (getActivity() != null) {
        getActivity().runOnUiThread(new Runnable() {
```

---

1292

Subscribe to updates at https://commonsware.com

Special Creative Commons BY-NC-SA 4.0 License Edition
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.

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 DEAD_EVENT 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))
    .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.
You can annotate a method with \texttt{@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., \texttt{RandomEvent}).

When an app calls \texttt{register()} on a \texttt{Bus}, Otto finds all of the \texttt{@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 \texttt{@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 \texttt{register()} processing.

So, if we had a \texttt{@Produce} method in the sample app that returned a \texttt{RandomEvent}, the \texttt{onRandomEvent()} method would have been invoked immediately with the result of calling that \texttt{@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 \texttt{@Produce} pattern would allow you to cache that result and give it to the new activity or fragment.

\textbf{greenrobot’s EventBus}

Of the three major in-process event bus implementations, \texttt{greenrobot’s EventBus} is slightly less popular than \texttt{LocalBroadcastManager} and Otto. \texttt{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 \texttt{ActionBarSherlock}, \texttt{ViewPagerIndicator}, and many others).

This is not to say that \texttt{greenrobot’s EventBus} is weaker than the alternatives. In fact, it may be the most powerful of the three. Like Otto, \texttt{greenrobot’s EventBus} has Guava’s \texttt{EventBus} in its heritage. Unlike Otto, \texttt{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.
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:

```java
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 de.greenrobot.event.EventBus;

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, 0);
    }

    @Override
    protected void doWakefulWork(Intent intent) {
        EventBus.getDefault().post(new RandomEvent(rng.nextInt()));
    }

    @Override
    public void onDestroy() {
        EventBus.getDefault().unregister(this);

        super.onDestroy();
    }
}
```
public void onEvent(RandomEvent 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.stringnotif_title))
        .setContentText(Integer.toHexString(event.value))
        .setSmallIcon(android.R.drawable.stat_notify_more)
        .setTicker(getString(R.stringnotif_title));

    PendingIntent contentIntent = 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 event handler for the RandomEvent, by calling register() on the EventBus instance in onCreate(). Version 2.2.0 of greenrobot’s EventBus added ordered event handling, modeled after the ordered broadcasts supplied by Android. The two-parameter form of register() takes a priority as the second parameter, with higher numbers indicating higher priority.

The service itself then has an onEvent() method for our RandomEvent, 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.
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 onEvent():

```java
public void onEvent(final RandomEvent event) {
    getActivity().runOnUiThread(new Runnable() {
        @Override
        public void run() {
            adapter.add(event);
        }
    });

    EventBus.getDefault().cancelEventDelivery(event);
}
```

The default behavior of threading with greenrobot’s EventBus is to deliver the event on the same thread that raised the event, much like Otto and LocalBroadcastManager. Hence, we use runOnUiThread() to arrange to update the ArrayAdapter on the main application thread, as our event is raised by the background thread from our WakefulIntentService.

Our onEvent() method also calls cancelEventDelivery(), which is analogous to abortBroadcast() with ordered broadcasts. This prevents lower-priority event handlers from receiving the event.

This works, because we registered this event handler with a priority of 1, higher than the 0 we used with the service:

```java
@Override
public void onResume() {
    super.onResume();

    EventBus.getDefault().register(this, 1);
}

@Override
public void onPause() {
    EventBus.getDefault().unregister(this);

    super.onPause();
}
```

Hence, while the fragment is in the foreground, it will handle the RandomEvent; otherwise, the service will.
Other Notable Capabilities

The biggest change you can (sometimes) utilize with greenrobot’s EventBus is to have events delivered to alternative threads. This is denoted by adding a suffix to the onEvent() method name. For example, having a MainThread suffix on the method name indicates to greenrobot’s EventBus that we want this method to be called on the process’ main application thread. With Otto, the only thread we can be called on is the thread used for posting the event. That is the default behavior with greenrobot’s EventBus too, so the onEvent() method in ScheduledService is supposed to be called on the thread where we used post(). However, the suffix system allows you to designate another thread. There are three possible suffix values:

- MainThread, to deliver the event on the main application thread
- BackgroundThread, which will use the current thread if it is not the main application thread, otherwise will use a separate thread
- Async, which will always use a separate thread

If our fragment’s method were named onEventMainThread(), we would get our RandomEvent on the main application thread, and we can add the event to the EventLogAdapter directly, avoiding the extra Runnable and runOnUiThread() logic that is used currently.

However, the ordered event system has a key limitation: it only works with the “post thread” (i.e., the default behavior of onEvent() with no suffix). Since we are using the ordered event system, we cannot take advantage of greenrobot’s alternative thread delivery system.

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.
- 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
with an `onEvent(AppEvent)` method to find out about all of your events that inherit from `AppEvent`. 
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
**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
HOME SCREEN APP WIDGETS

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
<?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">
```
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...
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
catch 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:

```xml
<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:

```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:
HOME SCREEN APP WIDGETS

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
<?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"/>
</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:

```java
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,
```
HOME SCREEN APP WIDGETS

```java
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
HOME SCREEN APP WIDGETS

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:
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.
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. onDelete() 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 onDelete() 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 onDelete() 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 Pair0fDice 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 Pair0fDice 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)
HOME SCREEN APP WIDGETS

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
<?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"
```
This layout is then the one you would reference from your app widget metadata, to be used when the app widget is first created:

```xml
<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.
HOME SCREEN APP WIDGETS

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 414: API Demos App Widget, Resizing](image)

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.
HOME SCREEN APP WIDGETS

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
<?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"/>

</RelativeLayout>
</RelativeLayout>

Our widget_provider.xml resource stipulates our desired android:resizeMode and minimum resize dimensions:

```xml
<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
HOME SCREEN APP.Widgets

&lt;intent-filter&gt; of our &lt;receiver&gt; in the manifest, indicating that we want APPWIDGET_OPTIONS_CHANGED as well as ACTION_UPDATE:

```xml
&lt;application
  android:allowBackup="false"
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name">
&lt;receiver
  android:name="AppWidget"
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name">
&lt;intent-filter&gt;
  &lt;action android:name="android.appwidget.action.APPWIDGET_UPDATE"/&gt;
  &lt;action android:name="android.appwidget.action.APPWIDGET_OPTIONS_CHANGED"/&gt;
&lt;/intent-filter&gt;
</receiver>
&lt;/receiver&gt;
&lt;/application&gt;
```

Then, our app widget implementation can override an onAppWidgetOptionsChanged() method:

```java
@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
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:
When the user resizes our app widget, we show the new size ranges:
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:

```java
// 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);
    }
}
```
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:
HOME SCREEN APP WIDGETS

Our lockscreen layout simply adds a third die, middle_die:

```xml
<?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
HOME SCREEN APP WIDGETS

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:

```java
@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:

```java
@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.setOnClickListenerPendingIntent(R.id.background, pi);
```
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:
Tapping the “+” indicator (and, if needed, entering your device PIN or password), brings up an app widget chooser:
Choosing `TwoOrThreeDice` will then add the app widget to the lockscreen, with three dice, not two:
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:
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:

![Figure 420: The XOOM tablet’s app widget gallery](image)

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:
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.
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:
Figure 422: 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.
Let's take a look, in the form of the AppWidget/LoremWidget sample project, which will put a ListView of 25 nonsense 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:

```java
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);

            widget.setViewVisibility(R.id.words, View.GONE);
        }

        appWidgetManager.updateAppWidget(appWidgetIds, widget);
        appWidgetManager.notifyAppWidgetViewDataChanged(appWidgetIds, widget);
    }
}
```

---

**Subscribe to updates at https://commonsware.com**

**Special Creative Commons BY-NC-SA 4.0 License Edition**
PendingIntent clickPI=PendingIntent
   .getActivity(ctx, 0,
            clickIntent,
            PendingIntent.FLAG_UPDATE_CURRENT);

    widget.setPendingIntentTemplate(R.id.words, clickPI);

    appWidgetManager.updateAppWidget(appWidgetIds[i], widget);
}

super.onUpdate(ctx, 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.
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:

```java
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 LoremViewsFactory 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
<?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"/>
</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.
For example, here is LoremViewsFactory, the one used by the LoremWidget project:

```java
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]);
        return row;
    }
}
```

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.
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
<?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
-->
<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 nonsense words into that `RemoteViews` for the `TextView` inside it. It also creates an `Intent` and puts the nonsense word in as an `EXTRA_WORD` extra, then provides that `Intent` to `setOnClickFillInIntent()`. In addition, it adds the app widget instance ID as an
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:

```xml
<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 423: The preview of LoremWidget](image)

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
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:

```java
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.
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 424: A XOOM home screen, showing the LoremWidget on the left](image)

The ListView is live and can be scrolled. Tapping an entry brings up the corresponding Toast:
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:
The user can then drag those handles to expand or shrink the app widget in the specified dimensions:
Figure 427: The resized LoremWidget
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
read data represented by the Uri, update that data, or delete the instance outright. Or, given an 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
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
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:

```java
@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:

```java
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)
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)

```java
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
@SuppressWarnings("deprecation")
@Override
public void onPostExecute(Void arg0) {
    SimpleCursorAdapter adapter;
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        adapter = new SimpleCursorAdapter(
            ConstantsBrowser.this,
```
If you need more control over the views than you can reasonably achieve with the stock view construction logic, subclass `SimpleCursorAdapter` and override `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 old 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, being 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:

```java
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();
}
```

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`:

```java
@Override
public void onClick(DialogInterface dialog, int which) {
    ContentValues values = new ContentValues(2);
    AlertDialog dialog = (AlertDialog)dialog;
    EditText title = (EditText)dialog.findViewById(R.id.title);
    EditText value = (EditText)dialog.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:

```java
@Override
protected Void doInBackground(ContentValues... values) {
```

---

**CONTENT PROVIDER THEORY**

---

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

```java
@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:

- 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 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.

**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 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.

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.
CONTENT PROVIDER THEORY

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:

```xml
<manifest version="1.0" encoding="utf-8">  
  <application>
    <provider>
      <activity>
        <intent-filter>
          <action android:name="android.intent.action.MAIN"/>
          <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
        <provider>  
          android:name="com.commonsware.android.constants.Provider"  
          android:authorities="com.commonsware.android.constants.Provider"  
          android:exported="false"/>  
        </activity>
      </provider>
    </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 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.

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>`. 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.
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().
onCreate()

Here is the onCreate() method for Provider, from the ContentProvider/ConstantsPlus sample application:

```java
@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:

```java
package com.commonsware.android.constants;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
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);
            }
        } finally {
            c.close();
        }
    }
}
```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
  // Upgrade code here...
}

finally {
  c.close();
}


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:

```java
@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);
    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:

```java
@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.
**update()**

Here is `update()` from `Provider`:

```java
@override
public int update(Uri url, ContentValues values, String where,
                   String[] whereArgs) {
    int count =
        db.getWritableDatabase()
          .update(TABLE, values, where, whereArgs);
    return getContent().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`:

```java
@override
public int delete(Uri url, String where, String[] whereArgs) {
    int count =
        db.getWritableDatabase().delete(TABLE, where, whereArgs);
    return getContent().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`:

```java
@override
public String getType(Uri url) {
```
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";
}
```
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
<?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 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/).

```java
@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);
        }
    }
    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:

```java
@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, ParcelFileDescriptor.MODE_READ_ONLY);
    }
    throw new FileNotFoundException(uri.getPath());
}
```

Here, we ignore the supplied mode parameter, treating this as a read-only file. That is safe in this case, since our only planned use of the provider is to serve read-only content to a WebView widget. If we wanted read-write access, we would need to convert the mode to something usable by the open() method on ParcelFileDescriptor.

**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:

```java
@Override
protected long getDataLength(Uri uri) {
    File f = new File(getContext().getFilesDir(), uri.getPath());
    return (f.length());}
```

### 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()`:

```java
@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:

```java
@Override
public Uri insert(Uri uri, ContentValues initialValues) {
    throw new RuntimeException("Operation not supported");
}

@Override
public int update(Uri uri, ContentValues values, String where, 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:

```java
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();

    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
dataLength() 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()`

Abstract `FileProvider` also has a convenience `copy()` static method that copies an `InputStream` to a `File`, used from the `FileProvider` `onCreate()` method:

```java
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
<?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"
            ...
    </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:

```java
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:

```java
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.

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:

```xml
<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.
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:

```java
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 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.

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:

```java
@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.

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:

```java
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.
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 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:
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 `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
<?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:

```xml
<provider
  android:name="android.support.v4.content.FileProvider"
  android:authorities="com.commonsware.android.cp.v4file"
  android:exported="false"
  android:grantUriPermissions="true"
  android:readPermission="com.commonsware.android.cp.v4file.READ">
  <meta-data
    android:name="android.support.FILE_PROVIDER_PATHS"
    android:resource="/xml/provider_paths"/>
</provider>
```

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 `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 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:

```java
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];
   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 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.
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
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 requeryed, 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.
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.
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
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:

```java
@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`:

```java
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
    return new CursorLoader(this, Provider.Constants.CONTENT_URI, 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
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:

```java
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:

```java
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
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](https://commonsware.com), “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.
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.
THE LOADER FRAMEWORK

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.
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
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
THE CONTACTS CONTRACT PROVIDER

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
<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:

```java
private static Uri CONTENT_URI=null;

static {
    int sdk=new Integer(Build.VERSION.SDK).intValue();
    if (sdk>=5) {
```
try {
    Class<?<>
class = Class.forName("android.provider.ContactsContract$Contacts");
    CONTENT_URI = (Uri)class.getField("CONTENT_URI").get(class);
} 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.
**The Contacts Contract Provider**

*Figure 428: The PickDemo sample application, as initially launched*

*Figure 429: The same application, after clicking the “Gimme!” button, showing the list of available people*
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.
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
<?xml version="1.0" encoding="utf-8"?>
<manifest
    android:versionCode="1"
    android:versionName="1.0"
    package="com.commonsware.android.contacts.spiners"
    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"
        android:name=".ContactSpinners">
        <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
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 ContactsDemo activity is simply a ListActivity, though it sports a Spinner to go along with the obligatory ListView:

```xml
<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:

```java
package com.commonsware.android.contacts.spiners;

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);
    }
```
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.
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:

```java
package com.commonsware.android.contacts.spiners;

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
            }));
    }
}
```
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:

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
                                         }));
    }
}
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.spiners;

import android.app.Activity;
import android.os.Build;
import android.widget.ListAdapter;
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 final 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 ContactsDemo activity, since that is the default perspective:
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:
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.
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 each for the person’s first name and phone number:

```xml
<?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:
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:

```java
package com.commonsware.android.inserter;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.provider.ContactsContract.
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);
        }
    }
}
```
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 435: The add-or-edit-a-contact activity](image)

... where if the user chooses “Create new contact”, they are taken to the ordinary add-a-contact activity, with our data pre-filled in:
Figure 436: 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 ContactInserter activity, and a new phone number will be added from those Intent extras.
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.
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).
THE CALENDARCONTRACT PROVIDER

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
<manifest
package="com.commonsware.android.cal.query"
android:versionCode="1"
android:versionName="1.0">
<uses-sdk android:minSdkVersion="14"/>
</manifest>
```
The Calendar Contract Provider

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"/>

    </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:

```java
@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:

```java
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:

```java
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
    return new CursorLoader(this, CalendarContract.Events.CONTENT_URI,
                             PROJECTION, null, null,
```
The CalendarContract Provider

We query on CalendarContract.Events.CONTENT_URI, asking for a certain set of columns indicated by our PROJECTION static data member:

```java
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():

```java
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:

```java
@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);
    }
```
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:

![Calendar events](image)

*Figure 437: 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](https://commonsware.com) (CDD):
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.
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.
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)
THE MEDIASTORE PROVIDER

- 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 StackOverflow 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.
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.

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:

```java
@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:

```xml
<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">

    <com.loopj.android.image.SmartImageView
        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>
```
We will cover the com.loopj.android.image.SmartImageView in the next section; for the moment, take it on faith that it will be an image, eventually populated by a thumbnail of the video for that row.

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:

```java
@Override
public void onActivityCreated(Bundle state) {
    super.onActivityCreated(state);

    String[] from =
    int[] to = { android.R.id.text1, R.id.thumbnail };
    SimpleCursorAdapter adapter =
        new SimpleCursorAdapter(getActivity(), R.layout.row, null,
                                  from, to, 0);

    adapter.setViewBinder(new ThumbnailBinder());
    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):

```java
@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. You might think that we would use Picasso for this, having covered it back in the chapter on Internet access and having noted that it handles loading from local sources as well as Internet ones. However, Picasso does not support loading video thumbnails directly, and it is not designed with extensibility in mind.

So, instead, we will turn to another solution.

**Attaching the ViewBinder**

First, 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 setValue() 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.

**Integrating the SmartImageView**

Also, as seen earlier in this section, our row.xml layout resource refers to a com.loopj.android.image.SmartImageView widget. This comes from a third-party JAR, written by James Smith. SmartImageView handles the work of populating an ImageView asynchronously from some data source. It has built-in support for downloading images from a URL or loading from contacts, much like Picasso. And, compared to Picasso, SmartImageView is not all that smart, as Picasso has many more capabilities.

However, SmartImageView is easily extensible to support obtaining images from arbitrary sources, in stark contrast to the “directly hack the Picasso source code in cryptic ways” alternative.
Defining the VideoThumbnailImage

That extensibility model comes in the form of a SmartImage interface. SmartImageView knows how to populate itself from a SmartImage, where the getBitmap() method of SmartImage is called on a background thread, where it can take some time loading in the data.

So, we define a VideoThumbnailImage implementation of SmartImage as follows:

```java
private static class VideoThumbnailImage implements SmartImage {
    private int videoId;
    private int thumbnailKind;

    VideoThumbnailImage(int videoId, int thumbnailKind) {
        this.videoId = videoId;
        this.thumbnailKind = thumbnailKind;
    }

    @Override
    public Bitmap getBitmap(Context ctxt) {
        return MediaStore.Video.Thumbnails.getThumbnail(ctxt.getContentResolver(),
            videoId, thumbnailKind,
            null);
    }
}
```

Our constructor takes two parameters:

- the ID of the video for which we need a thumbnail, which will come from our Cursor of retrieved videos
- the type of thumbnail that we want, which needs to be either MediaStore.Video.Thumbnails.MICRO_KIND or MediaStore.Video.Thumbnails.MINI_KIND

For ListView rows, MICRO_KIND will usually be more suitable, as it is smaller (96 x 96, as opposed to MINI_KIND at 512 x 384).

In our getBitmap() implementation, we call the static getThumbnail() method on MediaStore.Video.Thumbnails. This takes our video ID, our thumbnail kind, a ContentResolver (obtained from the passed-in Context), and an optional BitmapFactory.Options (here, set to null).
**THE MEDIASTORE PROVIDER**

A production-grade implementation of this might well use that BitmapFactory.Options and its inBitmap parameter to reuse an existing Bitmap object from a pool, to help minimize heap fragmentation.

getThumbnail() may be quick, or it may not, but since getBitmap() is called on a background thread, the time required to retrieve the thumbnail is less of an issue.

**Populating the SmartImageView**

Our ThumbnailBinder then uses the VideoThumbnailImage in its setViewValue() method:

```java
private static class ThumbnailBinder implements SimpleCursorAdapter.ViewBinder {
    @Override
    public boolean setViewValue(View v, Cursor c, int column) {
        if (column == c.getColumnIndex(MediaStore.Video.Media._ID)) {
            VideoThumbnailImage thumb = new VideoThumbnailImage(
                c.getInt(column),
                MediaStore.Video.Thumbnails.MICRO_KIND);

            ((SmartImageView)v).setImage(thumb,
                R.drawable.ic_media_video_poster);

            return(true);
        }

        return(false);
    }
}
```

If our ViewBinder is being asked to bind the video ID column (MediaStore.Video.Media._ID), we create an instance of VideoThumbnailImage using that ID, and attach it to the SmartImageView via setImage(). We are passed in that SmartImageView, but it is supplied as a generic View object, so we are forced to cast it to SmartImageView. setImage() also takes a placeholder drawable resource, to be used while the real image is being fetched on a background thread.

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

```java
@Override
public void onListItemClick(ListView l, View v, int position, long id) {
    CursorAdapter adapter = (CursorAdapter) getListAdapter();
    Cursor c = (Cursor) adapter.getItem(position);
    int uriColumn = c.getColumnIndex(MediaStore.Video.Media.DATA);
    int mimeTypeColumn = c.getColumnIndex(MediaStore.Video.Media.MIME_TYPE);
    getContract().onVideoSelected(c.getString(uriColumn),
                                  c.getString(mimeTypeColumn));
}
```

The main activity — surprisingly named MainActivity — loads up a VideosFragment as a static fragment via the res/layout/main.xml resource:

```xml
<?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:

```java
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);
    }
}

The Results

Running this on a device with videos available should show the list of those videos, complete with title and thumbnail:
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.
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.
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)
Obtaining SQLCipher

SQLCipher for Android is available from its GitHub repository. The downloads area contains ZIP archives of what you need. As of February 2014, the current shipping version was 3.0.2.

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.

Employing SQLCipher

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).

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.

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, 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:

```java
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):

```java
@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 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):

```java
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:

```java
@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:

```java
@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, 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 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.

**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](https://commonsware.com) sample app, 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:

```java
static void encrypt(Context ctxt) {
    SQLiteDatabase.loadLibs(ctxt);

    File dbFile=ctxt.getDatabasePath(DATABASE_NAME);
    File legacyFile=ctxt.getDatabasePath(LEGACY_DATABASE_NAME);
```

---

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

```java
if (!dbFile.exists() && legacyFile.exists()) {
    SQLiteDatabase db = SQLiteDatabase.openOrCreateDatabase(legacyFile, "", null);
    db.rawQuery(String.format("ATTACH DATABASE '%s' AS encrypted KEY '%s';", 
            dbFile.getAbsolutePath(), PASSPHRASE));
    db.rawQuery("SELECT sqlcipher_export('encrypted')");
    db.execSQL("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 SQLiteCipher for Android by calling loadLibs() on the SQLiteCipher 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 SQLiteCipher for Android, using "" as the passphrase.
Encrypted Storage

- 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 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`:

```java
@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.

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:

```java
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

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:

```java
public class SQLCipherV3Hook implements SQLiteDatabaseHook {
    private static final String PREFERENCES =
        "net.sqlcipher.database.SQLCipherV3Helper";

    public static void resetMigrationFlag(Context ctxt, String dbPath) {
        SharedPreferences prefs =
            ctxt.getSharedPreferences(PREFERENCES, 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 =
            database.getSharedPreference(PREFERENCES, 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 SQLiteDatabaseOpenHelper constructor as the fifth parameter, which will be used when SQLiteOpenHelper works with the underlying SQLiteDatabase.

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:
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](https://commonsware.com) 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.
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

```java
new SQLCipherStrategy(getContext(), NAME, "atestpassword", LoadPolicy.SYNC)
```
(here, NAME is the name of the set of preferences, "atcpassword" 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:

```java
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
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](https://commonsware.com). 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.
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.
that supports this pattern, such as Jeff Gilfelt's SQLiteAssetHelper, available via a
GitHub repository.

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 ZIP file with 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.

You might think that you would store the SQLite database in your project's assets/
directory, given the name of the SQLiteAssetHelper class. That is not quite how it
works. Your raw database will need to go somewhere else that can be version-
controlled but is not part of normal APK packaging (e.g., create a misc/ directory in
your project and put it there). Then, you need to:

1. Create an assets/databases/ directory in your project
2. Use a ZIP utility (command-line zip, WinZip, native OS ZIP archive
capability, etc.) to compress your database file and put it in assets/
databases/ under the proper name

The “proper name” for the ZIP file is your database's original name, with the .zip
extension. So, for example, a foo.db raw SQLite database file would need to be ZIP-
compressed and stored in assets/databases/foo.db.zip. Particularly if you are
using Ant, you might consider adding commands to your build script to
automatically do this compression (e.g., using Ant's <zip> task).
The reason for the ZIP compression comes from an Android limitation – assets that are compressed by the Android build tools have a file-size limitation (around 1MB). Hence, you need 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.zip 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:

```java
package com.commonsware.android.dbasset;

import android.content.Context;
import com.readystatesoftware.sqliteasset.SQLiteAssetHelper;

class DatabaseHelper extends SQLiteAssetHelper {
    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 getWriteableDatabase() 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.
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.
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.
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.
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
<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>
```
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);
```
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);
    }
}
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...
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:

![AudioDemo sample application](image)

*Figure 439: 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.
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
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.
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`:  

1473
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:
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.
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:

```xml
<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 `OnCheckedChangeListener`, to find out when the user toggles the button:

```java
@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:

```java
@Override
public void onResume() {
    super.onResume();

    recorder = new MediaRecorder();
    recorder.setOnErrorListener(this);
    recorder.setOnInfoListener(this);
}
```
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:

```java
@Override
public void onPause() {
    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());
        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(),
```
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:

```java
@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:
        msg = getString(R.string.max_filesize);
        break;
    }
    Log.wtf("Exception in stopping recorder", e);
    // can fail if start() failed for some reason
}
```

```java
recorder.reset();
```
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
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.
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:

```java
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.
AUDIO RECORDING

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:

```java
private static final String BASENAME="recording-stream.amr";
```

We also set our output format to RAW_AMR, and our encoder to AMR_NB:

```java
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 Google 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.
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.
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
<?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"/>
</LinearLayout>
```
VIDEO PLAYBACK

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:

```java
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 ctrl;

    @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());

            ctrl = new MediaController(this);
            ctrl.setMediaPlayer(video);
            video.setMediaController(ctrl);
            video.requestFocus();
            video.start();
        }
    }
}
```

Here, we:

1. Confirm that our video file exists on external storage
2. Tell the VideoView which file to play
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 441: The VideoDemo sample application, showing a Creative Commons-licensed video clip](image)

Tapping on the video will pop up the playback controls:
Figure 442: 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>):
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.
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)
• 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:

```xml
<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:

```java
@override
public void onCreate(Bundle savedInstanceState) {
```

1492
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.
First, you have no control over the camera app itself. You do not even really know what app it is. You cannot dictat 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 to 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:

```java
public void doScan(View v) {
    (new IntentIntegrator(this)).initiateScan();
}
```

---

**USING THE CAMERA VIA 3RD-PARTY APPS**

Subscribe to updates at https://commonsware.com  
Special Creative Commons BY-NC-SA 4.0 License Edition
USING THE CAMERA VIA 3RD-PARTY APPS

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:

```java
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
• 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);
    }
```
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.
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.

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.

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.

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 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()](http://developer.android.com/reference/android/hardware/Camera.html#getCameraInfo(int, android.hardware.Camera.CameraInfo)) 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.

**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.
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.

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 a 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 VerticalSeekBar to control zoom levels, so you can see how this is used.

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.
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
WORKING DIRECTLY WITH THE CAMERA

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.

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:

```xml
<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.
WORKING DIRECTLY WITH THE CAMERA

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:

```xml
<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.

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">

    "Main 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>

    <activity
        android:name="FullScreenActivity"

    <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.

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.

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.

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():

---

WORKING DIRECTLY WITH THE CAMERA

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.

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():

---

1513
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 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:

```java
@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();
                }
            } catch (Exception e) {
                getHost().onCameraFail(FailureReason.UNKNOWN);
            }
        } else {
            getHost().onCameraFail(FailureReason.NO_CAMERAS_REPORTED);
        }
    }
    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);
    }
    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
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:

```java
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()`.

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.

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 `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;
import android.os.Bundle;
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 boolean singleShotProcessing = false;
    private SeekBar zoom = null;
    private long lastFaceToast = 0;
    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());
    }

    @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);
        zoom.setSeekBarChangeListener(this);

        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);

    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();
        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:

    1522
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:
    item.setChecked(!item.isChecked());

} 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);
            }
        });
    }
}
@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()) {
        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;
        get.Activity().runOnUiThread(new Runnable() {
            @Override
            public void run() {
                takePictureItem.setEnabled(true);
            }
        });
        DisplayActivity.imageToShow=image;
        startActivity(new Intent(getActivity(), DisplayActivity.class));
    } else {
        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 {
        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;
        }
    }

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.

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.

**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.

```java
@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);
```

WORKING DIRECTLY WITH THE CAMERA
**WORKING DIRECTLY WITH THE CAMERA**

```java
recordItem = menu.findItem(R.id.record);
stopRecordItem = menu.findItem(R.id.stop);

if (isRecording()) {
    recordItem.setVisible(false);
    stopRecordItem.setVisible(true);
    takePictureItem.setVisible(false);
}
setRecordingItemVisibility();
```

record() on CameraFragment eventually routes to record() on CameraView, which goes through the aforementioned recipe:

```java
public void record() throws Exception {
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
        throw new UnsupportedOperationExce$$ion("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);
    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;
    }
}
```

1529
record() uses three hooks supplied to CameraHost for configuring the recording:

- configureRecorderAudio(), where SimpleCameraHost uses
  MediaRecorder.AudioSource.CAMCORDER:

```java
@Override
public void configureRecorderAudio(int cameraId, MediaRecorder recorder) {
    recorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
}
```

- configureRecorderProfile(), where SimpleCameraHost uses the
  QUALITY_HIGH profile:

```java
@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, CamcorderProfile.QUALITY_LOW));
    } else {
        throw new IllegalStateException("cannot find valid CamcorderProfile");
    }
}
```

- configureRecorderOutput(), where SimpleCameraHost calls
  getVideoPath() and uses that location for the output

```java
@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:

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()`.

**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
- 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 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.

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

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.

---

1536

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

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.
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.
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
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 port of certain MediaRouter support classes to work with the native API Level 11+ version of fragments and the action bar. These classes are a stop-gap measure, until Google eventually publishes their own version of those support classes.

Note that using CWAC-MediaRouter still requires the mediarouter Android library project, as the CWAC library only contains those classes that needed modification. Since the mediarouter library requires the appcompat library project and the support-v4 JAR, you will need all those dependencies, even if you are not using much from them. While this will add a few MB of space to your APK in debug mode, much of that extra stuff should get stripped out by ProGuard when you build the release version of your app.

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.

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 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, 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:

```xml
<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
<?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">
```
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;
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 onCreateOptionsMenuMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        MenuItem item = menu.findItem(R.id.route_provider);
        MediaRouteActionProvider provider =


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 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:
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:
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:
Here, we can adjust the volume, plus disconnect from the route. Disconnecting shows our `MediaRouteActionProvider` with the default white highlight:
The white highlight means that there are possible routes, though none in use. Tapping the icon brings up a connection dialog:

```java
MediaRouter.RouteInfo{ uniqueId=android/support.v7.media.SystemMediaRouteProvider:DEFAULT_ROUTE, name=Phone, description=null, enabled=true, connecting=false, playbackType=0, playbackStream=3, volumeHandling=1, volume=11, volumeMax=15, presentationDisplayId=-1, extras=null, providerPackage_name=android }
```
**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:
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:
The `MediaRouteActionProvider`, when tapped, will pop up a dialog of available routes that the user can select:

```java
MediaRouter.RouteInfo{ uniqueId=android/support.v7.media.SystemMediaRouteProvider:DEFAULT_ROUTE, name=Phone, description=null, enabled=true, connecting=false, playbackType=0, playbackStream=3, volumeHandling=1, volume=4, volumeMax=15, presentationDisplayId=-1, extras=null, providerPackageName=android }```

*Figure 450: MediaRouter ActionProvider Demo, on a Nexus 4, Showing ActionProvider*
MEDIA ROUTES

Figure 451: 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:
If the user chooses a route from the dialog, our `onRouteSelected()` method will be called to reflect the new selection:
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:
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:

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/route_provider"
        android:actionProviderClass="com.commonsware.cwac.mediarouter.MediaRouteActionProvider"
        android:showAsAction="always"
</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.
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:

```java
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:

```java
@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:

```java
@Override
public void onAttach(Activity host) {
```

1561
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

1562
As with most fragments, we inflate this layout in `onCreateView()`, holding onto the `TextView` and `ScrollView` widgets:

```java
@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:

```java
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:
As before, tapping on the “cast” action bar item pops up our dialog of available routes:

![Remote Playback Client Demo](image)
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:

```java
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();
    }
};
```
The connect() method handles connecting to the remote playback device and starting a session:

```java
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));
                    getActivity().supportInvalidateOptionsMenu();
                }
                @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.
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:

```java
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()`.

---

1567
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 supportInvalidateOptionsMenu(), to force the action bar to be repopulated.

### 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 onCreateOptionsMenu() implementation uses those, plus the existence of a non-null client, to configure the action bar items:

```java
@override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.main, menu);

    if (client != null) {
        if (isPlaying) {
            menu.findItem(R.id.stop).setVisible(true);
            if (isPaused) {
                menu.findItem(R.id.play).setVisible(true);
            } else {
                menu.findItem(R.id.pause).setVisible(true);
            }
        } else {
            menu.findItem(R.id.play).setVisible(true);
        }
    }

    MenuItem item=menu.findItem(R.id.route_provider);
    MediaRouteActionProvider provider=(MediaRouteActionProvider)MenuCompat.getActionProvider(item);
    provider.setRouteSelector(selector);
}
```

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 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:

*Figure 457: 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:

```java
@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);
    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:

```java
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;
            getActivity().supportInvalidateOptionsMenu();
        }
```
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

`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:
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.

And, indeed, that is what we do... except that we have to deal with a bug:

```java
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:

```java
class RunnableSessionActionCallback extends SessionActionCallback
    implements Runnable {
    @Override
    public void onResult(Bundle data, String sessionId,
                          MediaSessionStatus sessionStatus) {
        run();
    }
    @Override
    public void run() {
        transcript.removeCallbacks(this);
    }
}
```

StopCallback, as seen in the stop() method above, extends RunnableSessionActionCallback and overrides run():

```java
private class StopCallback extends RunnableSessionActionCallback {
    @Override
    public void run() {
        super.run();
    }
}
```

1573
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 run() implementation in StopCallback first chains to the superclass, which calls removeCallback(). This ensures that our Runnable will be invoked at most once, since onResult() forwards processing to run(). Hence, if onResult() is called before the timeout, we remove the Runnable, so it is not executed twice. Alas, this workaround does not presently handle the case where onResult() is called after the Runnable has run — this will be addressed in the future, hopefully by simply ripping out this hack, once the underlying bug is fixed.

The run() implementation in StopCallback also updates our flags, refreshes the action bar, and logs a message to the transcript. The result will look like:

![Remote Playback Demo](https://commonsware.com/Android/figures/remoteplaybackclientdemo.png)

*Figure 459: 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`:

```java
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`:

```java
private class PauseCallback extends RunnableSessionActionCallback {
    @Override
    public void run() {
        super.run();
        isPaused=true;
        getActivity().supportInvalidateOptionsMenu();
        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:

```java
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
    public void run() {
        super.run();
        isPaused = false;
        getActivity().supportInvalidateOptionsMenu();
        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:

private void disconnect() {
    isPlaying = false;
    isPaused = false;

    if (client != null) {
        logToTranscript(getActivity().getString(R.string.session_ending));
        EndSessionCallback endCB = new EndSessionCallback();

        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. To be sure we get 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:

```java
private class EndSessionCallback extends RunnableSessionActionCallback {
  @Override
  public void run() {
    super.run();
    client.release();
    client = null;

    if (getActivity() != null) {
      getActivity().supportInvalidateOptionsMenu();
      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 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.
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 Google 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 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.
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:
Tapping that will give you various options for what secondary display to emulate:
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:
Normaly, 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, making it somewhat of a challenge to work with the upper-left corner of your app.

In practice, before you ship a Presentation-capable app, you will want to test it with an actual physical external display.
SUPPORTING EXTERNAL DISPLAYS

**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.
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:

![Wireless Display Settings](image)

*Figure 463: 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.
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 DISPLAYCATEGORY_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 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:

```java
@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.
Supporting External Displays

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:

```java
@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():

```java
@TargetApi(Build.VERSION_CODES.JELLY_BEAN)
private class RouteCallback extends SimpleCallback {
    @Override
    public void onRoutePresentationDisplayChanged(MediaRouter router, RouteInfo route) {
```

1589
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:

```java
@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
            }
        }
    }
}
```
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:

```java
@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:

```java
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
private void clearPreso() {
    if (preso != null) {
        preso.dismiss();
        preso=null;
    }
}
```
Our onPause() uses clearPreso() and removeCallback() to unwind everything:

```java
@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:

![Android app interface](image)

You should see a Web page on the secondary display!

*Figure 464: Nexus 10, No Emulated Secondary Display, Showing Sample App*

If you run this *with* an external display, the external display will show our WebView:
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
SUPPORTING EXTERNAL DISPLAYS

sample creates an instance of PresentationHelper in onCreate(), stashing the object in a data member:

```java
@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:

```java
@Override
public void onResume() {
    super.onResume();
    helper.onResume();
}
```

```java
@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:

```java
@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
activity itself), or false if the activity is moving to the background (triggered via onPause()):

```java
@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:

```java
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():

```java
public void onResume() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
        handleRoute();
        mgr.registerDisplayListener(this, null);
    }
```
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:

```java
@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:

```java
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) {
```

1596
SUPPORTING EXTERNAL DISPLAYS

```java
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()`.

1597
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:

```java
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
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.
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().
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:

```java
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 getApplicationContext() {
        if (preso != null) {
            return(preso.getApplicationContext());
        }
        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.
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:

```java
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
}
```
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;
}
SUPPORTING EXTERNAL DISPLAYS

(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:

```java
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);
    WebView result = getWebview().loadUrl(getArguments().getString(ARG_URL));
    return (result);
  }
}
```
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:

```xml
<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:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    setContentView(R.layout.activity_main);
}
```
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():

```java
@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()) {
                showPreso(route);
                Log.d(getClass().getSimpleName(), "switched route");
            } else {
                // no-op: should already be set
            }
        } else {
            clearPreso(true);
            Log.d(getClass().getSimpleName(), "disabled route");
        }
    }
}
```
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:

```java
private void showPreso(RouteInfo route) {
    if (inline.getVisibility() == View.VISIBLE) {
        inline.setVisibility(View.GONE);
        prose.setText(R.string.secondary);
        Fragment f = getFragmentManager().findFragmentByIndex(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:

```java
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:

```java
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.
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 a 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.
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 467: PresentationSlidesDemo, Showing Overflow](image)

The Slides

The slides themselves are a series of 20 drawable resources (img0, img1, etc.), put into the res/drawable-xdpi/ 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.
SUPPORTING EXTERNAL DISPLAYS

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:

```java
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,
                                          ViewGroup.LayoutParams.MATCH_PARENT));

        return null;
    }

    public void destroyItem(ViewGroup container, int position, Object item) {
        container.removeView((ImageView)item);
    }

    public void getPageResource(int position) {
        return SLIDES[position];
    }

    public String getPageTitle(int position) {
        return TITLES[position];
    }

    public int getCount() {
        return SLIDES.length;
    }

    public boolean isViewFromObject(View view, Object object) {
        return ((ImageView)view)==(ImageView)object;
    }

    public void startUpdate(ViewGroup container) {
    }

    public void finishUpdate(ViewGroup container) {
    }

    public boolean hasView(int position) {
        return true;
    }

    public View instantiateView(ViewGroup container) {
        return new ImageView(container.getContext());
    }
}
```

1610
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 a external display via a Presentation. As with the preceding sample app, this one uses a PresentationFragment, here named SlidePresentationFragment:

```java
package com.commonsware.android.preso.slides;
import android.content.Context;
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;

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.
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:

```java
@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:

```java
@Override
public void onResume() {
    super.onResume();
    helper.onResume();
}

@Override
public void onPause() {
```

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:

```java
@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:

```java
@Override
public void onResume() {
    super.onResume();
    helper.onResume();
}

@Override
public void 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:

```java
@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:

```java
@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:

```java
@Override
public void onPageSelected(int position) {
    if (preso != null) {
        preso.setSlideContent(adapter.getPageResource(position));
    }
}
```
Offering an Action Bar

The activity also sets up the action bar with three items:

```xml
<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
    android:id="@+id/last"
    android:icon="@android:drawable/ic_media_next"
    android:showAsAction="always"
    android:title="@string/last"/>

  <item
    android:id="@+id/present"
    android:checkable="true"
    android:checked="true"
    android:showAsAction="never"
    android:title="@string/show_presentation"/>
</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.

```java
@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);

    return (true);
    }
```
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.
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.

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.
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.
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.
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 72op-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.
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.
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
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).
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).
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.
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.
What is the “Ten-Foot UI”?  

The “ten-foot UI” is not referring to a UI that is 3.048 meters high or $9.87789527 \times 10^{-17}$ 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.

Different platform developers have taken different approaches to dealing with overscan, with varying impacts upon the developer.
The “Ten-Foot UI”

The Google TV Approach

Some environments, like Google TV, as part of setup, will determine the safe viewing size for the television, by having the user calibrate the device based on test images. Hence, Google TV will not attempt to display something that the television is incapable of displaying (assuming proper setup). However, this does mean that while you will be thinking of 720p or 1080p resolution, you may not get all that space, and so you need to design your app to accommodate this.

One common problem encountered here is a background image. Developers have already been schooled to avoid full-screen backgrounds due to the wide range of resolutions available on handheld Android devices. Google TV just adds to the mix, where there are thousands upon thousands of possible actual resolutions, all minute changes from one another based upon what a particular television can handle. You will need to take this into account (e.g., put the background image on top of a solid field of color, where that solid color matches the dominant color from the edges of the image, then keep the background image at a fixed resolution and allow the solid fills on the edges take care of the overscan area).

The OUYA Approach

OUYA, like many game consoles, takes the opposite approach. OUYA totally ignores 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.

One possibility is to conditionally load a different layout when you detect that you are on an OUYA, where that layout handles the overscan. For example, let us suppose that you have res/layout/activity_main.xml that you wish to display,
and that layout resource assumes edge-to-edge screen usage. You could also have res/layout/activity_main_ouya.xml with something like this:

```xml
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:paddingLeft="@dimen/ouya_horizontal_padding"
  android:paddingRight="@dimen/ouya_horizontal_padding"
  android:paddingTop="@dimen/ouya_vertical_padding"
  android:paddingBottom="@dimen/ouya_vertical_padding">

  <include layout="@layout/activity_main"/>

</FrameLayout>
```

Here, we wrap the contents of activity_main.xml in a FrameLayout that uses padding to inset the activity_main.xml contents from the edges. In this case, the FrameLayout is using a pair of dimension resources, ouya_horizontal_padding and ouya_vertical_padding, hard-wired to appropriate values for a 1080p display:

```xml
<resources>
  <dimen name="ouya_horizontal_padding">96px</dimen>
  <dimen name="ouya_vertical_padding">54px</dimen>
</resources>
```

You could also craft some sort of OverscanLayout custom ViewGroup that could do the insets dynamically based on a percentage of screen resolution and use that instead.

The fact that overscan is not handled by the device, but rather by application code, will have other lingering impacts. For example, an action bar is an unsuitable UI construct on the OUYA, not only for perhaps being too small, but also because it is anchored to the actual top of the display output, and therefore may get cut off or otherwise mangled by overscan.

**The Fire TV Approach**

As is noted in the chapter on Fire TV, Amazon took yet another approach for dealing with overscan and variations in TV resolutions: scaling.

You, as a developer, render your app to a 1080p (1920x1080) resolution space, regardless of whether the actual screen is 1080p, lower (e.g., 720p), or higher (e.g., 4K). Any overscan adjustments are also taken care of at the Fire TV level.
simply pretend that everything is a 1080p screen, and the hardware takes care of any differences.

**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, Google 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 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
- 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 UI.

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, Google TV, Android HDMI sticks), you can use -notouch qualifiers on resource sets to provide different values for dimension resources.
THE “TEN-FOOT UI”

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

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, #E8E8E8) 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.

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
• Android “HDMI sticks” or other Android set-top boxes can be found on eBay, Alibaba, or elsewhere
• Chromecast is 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 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.
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
SSL

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.
SSL

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.
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:

```java
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)
SSL

OkHTTP

Square’s OkHttp works along the same lines:

```java
SSLContext ssl = SSLContext.getInstance("TLS")
ssl.init(null, builders, null);
OkHttpClient okHttpClient = new OkHttpClient();
okHttpClient.setSSLContext(ssl); // 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, StackOverflow 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!
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.
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:

```java
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:
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")
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:

```java
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
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.foobaz.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.
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.
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 MemorizingTrustManager that you could consider using. It has been around for a few years, with a slow-but-steady set of updates.
SSL

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.

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.

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”.

```java
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.
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 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 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
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
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

```xml
<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:

```xml
<activity
  android:name=".SekritApp"
  android:label="Top Sekrit"
```

1654

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

```
<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:
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.
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:

```xml
<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:

```xml
<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
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>
```
App B has a `<uses-permission>` element to declare to the user that it wishes to access components defended by that permission:

```xml
<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.
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,
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.
ADVANCED PERMISSIONS

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,
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:
• 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)

**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
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 AbstractCPPProxy subclass of ContentProvider that does the “heavy lifting”, and a CallLogProxy subclass of AbstractCPPProxy that handles some of the details of proxying the CallLog versus something else.

In onCreate() of the AbstractCPPProxy, 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:
@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:
This is called at the top of each ContentProvider method that we are proxying, such as `insert()`:

```java
@Override
public Uri insert(Uri uri, ContentValues values) {
    checkTainted();
    returngetContext().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.
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.
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:

![Users Screen in Settings](image)

*Figure 469: Users Screen in Settings*

Tapping the “Add user or profile” entry allows the primary user to set up another user or restricted profile:
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.
Restricted Profiles and UserManager

Figure 471: 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:
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:

Figure 472: Restricted Profile Configuration Screen in Settings
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:

---

**Figure 473: Location Sharing Restrictions**
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:

```java
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);
  }
}
```
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:

```xml
<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.
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:

![Image: 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.
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.

1678
# 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:

```xml
<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.
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:

```java
@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:
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:

```java
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:

```java
private RestrictionEntry buildChoiceRestriction(Context ctxt, Bundle current) {
    RestrictionEntry entry =
        new RestrictionEntry(RESTRICTION_CHOICE, current.getString(RESTRICTION_CHOICE));
```
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,
or optionally an `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:

![Restricted Profile, Showing App Settings Icon](image)

Tapping that settings icon will “unfold” and display the restrictions that you configured via the `RestrictionEntry` objects:
The primary user can then interact with your restrictions, toggling checkboxes and popping up the list dialogs:
Figure 478: Restricted Profile, Showing Choice Restriction

Figure 479: Restricted Profile, Showing Multi-Select Restriction
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`:

```java
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) getSupportFragmentManager().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:
package com.commonsware.android.profile;
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));
        }
    }
}
The result, when run on a restricted profile with restrictions placed upon our app, is to show those restrictions:

![App Restrictions Demo](image)

*Figure 480: 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.
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.
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.
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.
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
<?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>
                <!-- List of intents here -->
            </intent-filter>
        </activity>
    </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);

        startActivity(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);
        v.setOnTouchListener(this);
    }
}
```
v.setOnTouchListener(this); 
 
mgr = (WindowManager) getSystemService(WINDOW_SERVICE);

WindowManager.LayoutParams params = new WindowManager.LayoutParams(
   WindowManager.LayoutParams.MATCH_PARENT,
   WindowManager.LayoutParams.MATCH_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.)
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.
Detecting Potential Tapjacker

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.
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:

```java
package com.commonsware.android.tj.detect;

import android.app.ActivityManager;
import android.app.ListActivity;
import android.content.pm.PackageManager;
import android.content.pm.PackageInfo;
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);
            }
        }
    }
}
```

1697
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
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
<?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"/>
</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
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, Google 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.
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.
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](https://commonsware.com) 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
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
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 Intent. 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.
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?
• 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
<?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"/>
  </TableRow>
  <TextView android:id="@+id/subject"/>
</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:

```java
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:

```java
@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);
        }
    });
    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:

```java
package com.commonsware.android.signature.dump;

import android.annotation.TargetApi;
import android.content.pm.PackageInfo;
import android.os.Build;
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:
MISCELLANEOUS SECURITY TECHNIQUES

Figure 481: 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
null),
pkgInfo.packageName.replace('.', '_') + ".bin");
new WriteThread(output, raw).start();
}

1710

Subscribe to updates at https://commonsware.com

Special Creative Commons BY-NC-SA 4.0 License Edition


**MISCELLANEOUS SECURITY TECHNIQUES**

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:

```java
package com.commonsware.android.signature.dump;

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;
import com.actionbarsherlock.app.SherlockFragment;

public class SignatureFragment extends SherlockFragment {
  private DateFormat fmt = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss", Locale.US);

  @Override
  public View onCreateView(LayoutInflater inflator, 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
```java
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:
A self-signed production signing key will resemble:
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
command and binding patterns, where the only effective behavior for either of them is to log positive-sounding messages to LogCat:

```java
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>
```

1715
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:

```xml
<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:

```xml
<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>
```
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:

```java
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:

```java
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 {
```
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:

```java
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 StackOverflow answer, that compares the bytes of two streams, looking for a match:

```java
// 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;
```
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.
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.
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 Intents 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 Intents (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.
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:

```xml
<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
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
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.
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.
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.
Trail: Hardware and System Services
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.
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
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.
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 `getMyCurrentLocationNow()` 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:

```java
@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
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.
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.
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:

java.lang.IllegalArgumentException: no providers found for criteria
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.
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
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.
**Installing 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.

You will need to download the “Google Play services” package in your SDK Manager (see highlighted line):

![Android SDK Manager, Showing “Google Play services”](image)

Then, if you are an Eclipse user, 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.

**Attaching Google Play Services**

When you create a project that is to use the fused location provider, you will need to add a reference to the Play Services library project, either via the Eclipse project properties dialog or via the `android update lib-project` command. This is the same process that you use for adding other Android library projects, such as ActionBarSherlock.
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:

```java
-keep class * extends java.util.ListResourceBundle {
    protected Object[][] getContents();
}
```

It is unclear if this is strictly needed for the fused location provider, as the Play Services library is used for other things beyond that provider.

### 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:

```java
@override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (readyToGo()) {
        if (getSupportFragmentManager().findFragmentById(android.R.id.content) == null) {
            getSupportFragmentManager().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:

```java
protected boolean readyToGo() {
    int status = GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);

    if (status == ConnectionResult.SUCCESS) {
        return(true);
    } else if (GooglePlayServicesUtil.isUserRecoverableError(status)) {
        ErrorDialogFragment.newInstance(status)
            .show(getSupportFragmentManager(),
                 TAG_ERROR_DIALOG_FRAGMENT);
    } else {
```
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:

```java
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);
    ErrorDialogFragment result=new ErrorDialogFragment();
    result.setArguments(args);
    return(result);
  }
```

THE FUSED LOCATION PROVIDER

Toast.makeText(this, R.string.no_fused, Toast.LENGTH_LONG).show();
finish();
return(false);
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.

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. 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 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.

---

Special Creative Commons BY-NC-SA 4.0 License Edition
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 `LocationClient` in `onCreate()` of our `WeatherFragment`, holding onto the object in a client data member:

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setRetainInstance(true);

    template = get Activity().getString(R.string.url);
    client = new LocationClient(getActivity(), this, this);
}
```

The first parameter to the `LocationClient` constructor is a suitable `Context`, in this case our hosting activity. The second and third parameters are callback objects, that will be invoked when certain events occur in the life and times of the `LocationClient`. Specifically, the second parameter is an implementation of `PlayServicesClient.ConnectionCallbacks`, and the third parameter is an implementation of `PlayServicesClient.OnConnectionFailedListener`. In our case, `WeatherFragment` implements both of these interfaces.

Given that we have a `LocationClient`, 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`:

```java
@Override
public void onResume() {
    super.onResume();

    client.connect();
}

@Override
public void onPause() {
    getWebView().removeCallbacks(this);
}
```
THE FUSED LOCATION PROVIDER

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 GooglePlayServicesClient.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 ignores onDisconnected(), 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
public void onDisconnected() {
    // unused
}

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 GooglePlayServicesClient.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);
        }
    }
THE FUSED LOCATION PROVIDER

```java
anyLuck=true;
}
catch (IntentSender.SendIntentException e) {
    Log.e(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.

Finding the Current Location

Given all that setup, actually getting the location is almost anti-climactic.

To find the current location, given a connected LocationClient, just call getLastLocation(). 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:

```java
@Override
public void run() {
    Location loc=client.getLastLocation();

    if (loc == null) {
        getWebView().postDelayed(this, 1000);
    } else {
        FetchForecastTask task=new FetchForecastTask();
        task.execute(loc);
    }
}
```

1745

Subscribe to updates at https://commonsware.com

Special Creative Commons BY-NC-SA 4.0 License Edition
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`.

The rest of the sample is unchanged from the original, and the user experience is identical to the original (except perhaps getting a forecast a bit faster).

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.

**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.

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 three 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_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
THE FUSED LOCATION PROVIDER

- 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)

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()`.
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
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`:

```java
@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:

```java
public String getLocalIPAddress() throws SocketException {
    Enumeration<NetworkInterface> nics = NetworkInterface.getNetworkInterfaces();
```
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:
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 block, 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.
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
<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"
        android:layout_height="match_parent"/>
</LinearLayout>
```
WORKING WITH THE CLIPBOARD

In `onCreate()`, we get our hands on our `ClipboardManager` system service:

```java
@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:

```java
public void pickMusic(View v) {
    Intent i=new Intent(Intent.ACTION_GET_CONTENT);
    i.setType("audio/*");
    // Do something with the returned audio...
}
```
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:

![Select music track](image)

*Figure 487: 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:

```java
@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);
            }
        }
    }
}
```
WORKING WITH THE CLIPBOARD

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:

```java
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));
        }
    }
```

1756
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.
WORKING WITH THE CLIPBOARD

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:

```java
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));
  }
}
```
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
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
java.lang.IllegalStateException: beginBroadcast() called while already in a broadcast
```
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.
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.
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
<?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"
        />
    </LinearLayout>
</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:

```java
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:
However, the dialer you get from clicking the dial button is better, showing you the number you are about to dial:

![Figure 489: The Android Dialer activity, as launched from DialerDemo](image)
**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.
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
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:

```java
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.
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:

```java
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
<?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"
>
```
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 — this will be covered in greater detail elsewhere in this book.

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:
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 an instance suitable for the device the code is running on:

```java
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());
}

return(new NewContactsAdapterBridge());

The Android 2.x edition uses ContactsContract to find just the mobile numbers:

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:

class 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);
    }
}
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:
import android.widget.RadioGroup;
import android.widget.Spinner;

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.

## 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.
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](https://github.com/commonsware/Android-SmsMonitor) sample project implements a BroadcastReceiver for SMS_RECEIVED, one with slightly elevated priority:
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.
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:

```java
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.
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 BroadcastReceiver 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:

```xml
<!-- 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.
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:

```xml
<!-- 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:

```xml
<!-- 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" />
   </intent-filter>
</service>
```
WORKING WITH SMS

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.
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 URL.

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:
WORKING WITH SMS

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:

```java
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.
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.
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.
NFC

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
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.
NFC

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
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.
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,
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
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:

```xml
<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:

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    nfc=NfcAdapter.getDefaultAdapter(this);
}
```
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:

```java
@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
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()`:

```java
@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:

```xml
<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:

```java
@Override
protected void onNewIntent(Intent intent) {
    if (inWriteMode &&
        NfcAdapter.ACTION_TAG_DISCOVERED.equals(intent.getAction())) {
        Tag tag=intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
```
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.
First, we need the roster of prefixes, defined in URLTagger as a static data member cunningly named PREFIXES:

```java
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:", "btpp://",
 "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:

```java
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
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:

```java
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) {
```
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) {
        
    }
}
NFC

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 NdefFormattable aspect by calling get() on the NdefFormattable class. If the tag is not NDEF now but can be formatted as NDEF, this should give us an NdefFormattable 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 NdefFormattable, to put the NdefMessage on it, we first connect() to the tag, then format() it, supplying the message. NdefFormattable 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
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>:

```xml
<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
NFC

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
NFC

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_NFC_SHARING_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 the back-ported, ActionBarSherlock-friendly version of WebViewFragment used in various places in this book. 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()):

```java
@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:

```java
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:

```java
@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:
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:

```java
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:
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:

```java
@Override
public NdefMessage createNdefMessage(NfcEvent arg0) {
    NdefRecord uriRecord =
        new NdefRecord.Ntf_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:

```java
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 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
<?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="@style/Theme.Sherlock">
        <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>
</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.

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    beamFragment = (BeamFragment) getSupportFragmentManager().findFragmentById(android.R.id.content);
    if (beamFragment == null) {
        beamFragment = new BeamFragment();

        getSupportFragmentManager().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);
}

private void handleIntent(Intent i) {
    if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(i.getAction())) {
        Parcelable[] rawMsgs =
```
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 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 */*):

```java
@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:

```java
@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 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.
NFC

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:

![SecretAgentMan UI](image)

*Figure 491: The SecretAgentMan UI*

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:

```xml
<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:

```java
@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:

```java
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;
}

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:
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:

```java
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():

```java
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:

```java
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:

```java
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;

    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();
                    } 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);
                }
            } else {
                ndef.connect();
            }
        } 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();
        }
    }
}
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...

**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>: 
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 = 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
- 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);
    return true;
}
```
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:

```java
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:

```java
@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. That, in turn, requires us to implement onNdefPushComplete(), where we disable push mode via a call to setNdefPushMessageCallback() with a null listener:

```java
@Override
public void onNdefPushComplete(NfcEvent event) {
    nfc.setNdefPushMessageCallback(null, this);
}
```
NFC

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():

```java
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.

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.
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
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,
which itself is wrapped in a <device-admin> element. The range of possible policies is shown in the following sample XML metadata file:

```xml
<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:
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:

```java
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()).
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
<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:

```java
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;
```
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:
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:
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...
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:
You need to set up this app as a device administrator to be able to use it. Click the button to visit the Settings app to make this change. On the Security screen that should appear after pressing the button, look for a Device administrators entry, tap on it, then check Lockscreen Locker in the list.

Once that is complete, the Switch widgets become enabled and usable:
The device admin metadata for this app specifies that we want to control keyguard features:

```xml
<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:

```java
public class LockscreenAdminReceiver extends DeviceAdminReceiver {
}
```
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
<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"/>

</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:

```java
@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:

```java
@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:
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
onCheckedChangeListener(). 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);
        }

1834
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
una
available on this particular device. If it is inac
tive, 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:

```java
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);
        }
    }
}
```
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`:

```java
@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:
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:

```xml
<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:

```java
@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();
}
```
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.
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 WakeLocks 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.
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 496: The WakeLock Gap](image)

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()
execution, and hold onto that WakeLock until the work is completed by the IntentService:

![Diagram of WakeLock usage]

*Figure 497: 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.
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)
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 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.
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 “APs & 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 “APs & 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:
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. Hence, you 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.

**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:
**Push Notifications with GCM**

```xml
<permission
    android:name="com.commonsware.android.gcm.client.permission.C2D_MESSAGE"
    android:protectionLevel="signature"/>
```

You also need the corresponding `<uses-permission>` element:

```xml
<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:

```xml
<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.

`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

You will wind up with a <receiver> element reminiscent of the following:

```xml
<receiver
    android:name="GCMBroadcastReceiverCompat"
    android:permission="com.google.android.c2dm.permission.SEND">
    <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE"/>
    </intent-filter>
</receiver>
```
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 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.

**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".
PUSH NOTIFICATIONS WITH GCM

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 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:

```xml
<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="8"
    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"/>
  <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">
```

1853
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.

### 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:

```java
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;
```
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:

![Google Developers Console, Showing Project Number (a.k.a., Sender ID)](image)

**Figure 498: Google Developers Console, Showing Project Number (a.k.a., Sender ID)**

**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 UnsupportedOperation Exception if the device is not running at least API Level 8:
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
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
    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);
    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("Could 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
    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
    }
}
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:

```java
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 onClick(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:

```java
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 onDelete(Intent message);

    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)) {
            onDelete(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()`):

```java
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);
    }

    @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:

```java
private static void sendMessage(String apiKey, List<String> devices, Properties data) throws Exception {
    Sender sender=new Sender(apiKey);
    Message.Builder builder=new Message.Builder();
    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)));
            }
        }
    }
}
```
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 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:

```bash
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)

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 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.

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():

```java
// 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.

**Pre-Release Features**

Google I/O 2013’s announcement included several extensions to GCM that were profiled as being available, but in truth are only available to approved “trial partners” at this time.

**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.
PUSH NOTIFICATIONS WITH GCM

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 and sign up to be part of the program by visiting the CCS page in the GCM documentation.

Upstream Messages

If you have signed up for 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 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 programatically.

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.

**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.

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.

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.
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,
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.
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:
Basic Use of Sensors

@Override
public List<Sensor> getSensorList() {
    List<Sensor> unfiltered =
        new ArrayList<Sensor>(mgr.getSensorList(Sensor.TYPE_ALL));
    List<Sensor> result = new ArrayList<>();
    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.
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:

```java
@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()`:

```java
@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
```

1879
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.
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:

```java
@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.
Basic Use of Sensors

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:

```java
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);
    }
}
```
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) {
}
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 SherlockListFragment. 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:

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:
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:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mgr=(SensorManager) getSystemService(Context.SENSOR_SERVICE);
    log=(SensorLogFragment) getSupportFragmentManager().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:

```java
@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

```java
class SensorLogAdapter extends ArrayAdapter<Float[]>{
    public SensorLogAdapter(SensorLogFragment sensorLogFragment) {
        super(sensorLogFragment.getActivity(),
                android.R.layout.simple_list_item_1,
                new ArrayList<Float[]>());
    }
}
```

```java
@SuppressLint("DefaultLocale")
@Override
public View getView(int position, View convertView, ViewGroup parent) {
```
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:
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.
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:
OTHER SYSTEM SETTINGS AND SERVICES

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
<?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 500: The SettingsSetter application

The checklist itself is filled with a few BooleanSetting objects, which map a display name with a Settings.System key:

```java
static class BooleanSetting {
    String key;
}
```
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);
    }
}
Three such settings are put in the list:

```java
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:

```java
@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
<?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:

```java
menuActivities.put(R.id.app, Settings.ACTION_APPLICATION_SETTINGS);
menuActivities.put(R.id.security,
```
When an option menu is chosen, the corresponding activity is launched:

```java
@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.
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 aSeekBar per stream:
<table>
<thead>
<tr>
<th>Alarm</th>
<th>Music</th>
<th>Ring</th>
<th>System</th>
<th>Voice</th>
</tr>
</thead>
<tbody>
<tr>
<td>SeekBar</td>
<td>SeekBar</td>
<td>SeekBar</td>
<td>SeekBar</td>
<td>SeekBar</td>
</tr>
</tbody>
</table>
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:

```java
@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:
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
OTHER SYSTEM SETTINGS AND SERVICES

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
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:
DEALING WITH DIFFERENT HARDWARE

- **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:

```xml
<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
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 Google TV device:

```xml
<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.
DEALING WITH DIFFERENT HARDWARE

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:

```xml
<uses-feature
    android:name="android.hardware.telephony"
    android:required="false" />
```
Then, at runtime, you can call `hasSystemFeature("android.hardware.telephony")` on a `PackageManager` 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
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
RESPONDING TO URLs

to. For example, here is the AndroidManifest.xml file from the Introspection/URLHandler sample project:

```xml
<?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 screen sizes -->
<supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"/>

<!-- Application package settings -->
<application
    android:icon="@drawable/cw"
    android:label="@string/app_name">

    <activity
        android:name="URLHandler"
        android:label="@string/app_name">

        <intent-filter>

            <!-- Main action and category -->
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>

        </intent-filter>

        <intent-filter>

            <!-- Default and browsable action and category -->
            <action android:name="android.intent.action.VIEW"/>
            <category android:name="android.intent.category.DEFAULT"/>
            <category android:name="android.intent.category.BROWSABLE"/>

            <!-- Data filter for PDF files -->
            <data android:mimeType="application/pdf"/>
        </intent-filter>

        <intent-filter>

            <!-- Default and browsable action and category -->
            <action android:name="android.intent.action.VIEW"/>
            <category android:name="android.intent.category.DEFAULT"/>
            <category android:name="android.intent.category.BROWSABLE"/>

            <!-- Data filter for custom URL scheme -->
            <data
                android:host="www.this-so-does-not-exist.com"
                android:path="/something"
                android:scheme="http"/>
        </intent-filter>

        <intent-filter>

            <!-- Custom action and category -->
            <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>
```
Responding to URLs

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

Respsonding to MIME types makes complete sense... if we implement something designed to handle such a MIME type.
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:

```java
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);
    }
}
```
RESPONDING TO URLS

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
RESPONDING TO URLs

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.
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.
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:
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.
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.
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 queryBroadcast Receivers(), 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 getInstalled Packages() 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 getPackage Info() 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.
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.
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:
• 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

1918
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
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
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.).
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.
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
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
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:

```java
private ComponentName pluginCN = null;
```

In onResume(), if we do not have a plugin yet, we send the broadcast to try to find one:

```java
@override
public void onResume() {
```

1925
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);
        }
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`:

At this point, we wait for the user to click the Refresh options menu item.
Requesting RemoteViews

When the user does indeed choose Refresh, we call a `refreshPlugin()` method on the host activity:

```java
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 `pluginBroadcastReceiver` 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.
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:

```java
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
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.android.rv.plugin"
    android:minSdkVersion="7"
    android:versionCode="1"
    android:versionName="1.0">
    <uses-permission android:name="com.commonsware.android.rv.host.ACT_AS_PLUGIN"/>
</manifest>
```
In res/values/bools.xml, we define that boolean resource to be false, meaning the activity will not appear in the launcher:

```xml
<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:

```xml
<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).
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:

```xml
<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:

```xml
<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:

```xml
<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:

```java
registerReceiver(plugin, pluginFilter, PERM_ACT_AS_PLUGIN, null);
```

That permission is defined in a static data member:

```java
public static final String PERM_ACT_AS PLUGIN= "com.commonsware.android.rv.host.ACT_AS PLUGIN";
```
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.
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.
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:
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).
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`:

```java
package com.commonsware.android.cpproxy.provider;

import android.content.ContentUris;
import android.content.ContentUris;
import android.net.Uri;
import android.net.Uri;
import android.provider.CallLog;
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`:

```java
private static final String PREFS_TAINTED="tainted";
private boolean tainted=false;

public AbstractCPProxy() {
    super();
}

@Override
public boolean onCreate() {
    SharedPreferences prefs=
        PreferenceManager.getDefaultSharedPreferences(getContext());
    return(true);
}
```

The resulting `CrossProcessCursorWrapper`, as originally shown in a StackOverflow answer, looks like this:
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
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()).
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
<?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"/>

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name">

        <activity
            android:name=".CPPProxyConsumerActivity"
            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:

```java
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:
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 READCALENDAR 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 READCONTACTS requirement in a plugin, so users could elect to install the plugin or not, and the main app would not need to request READCONTACTS 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.
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.
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:

```java
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);
    }
}
```
In onCreate(), we:

```java
package Manager Tricks

@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));
    }
}
```
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:
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.

**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.

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()`:

```java
@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:

```java
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 StackOverflow answer to help with this. The IntentFilterAdapter and helper code looks like this:

```java
// 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;
            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('|');
                }

                buf.append(type);
            }
        }

        if (filter.countDataSchemes() > 0) {
            buf.append(" : ");
        }
    }
}
buf.append(filter.getDataScheme(0));

if (filter.countDataSchemes() > 1) {
    buf.append(" (other schemes)\n");
}

if (filter.countDataPaths() > 0) {
    buf.append(" : ");
    buf.append(filter.getDataPath(0));

    if (filter.countDataPaths() > 1) {
        buf.append(" (other paths)\n");
    }
}

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:

![Preferred Activities Demo](image)

*Figure 503: 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:

```java
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;

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.

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 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
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
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 504: The Android local search popup, showing the IME and a previous search
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.
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
<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:
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.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;
    private 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());
    }
}
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 aListAdapter for the whole word list:

```java
class LoremDemo extends AppCompatActivity {

    private static final int LOCAL_SEARCH_ID = R.id.local_search;
    private static final int GLOBAL_SEARCH_ID = R.id.global_search;

    private SearchManager searchManager;
    private SearchView searchView;
    private ArrayAdapter adapter;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lorem);

        searchManager = (SearchManager) getSystemService(SEARCH_SERVICE);
        searchView = (SearchView) findViewById(R.id.search);
        searchView.setIconifiedByDefault(true);
        searchView.setSearchableInfo(new SearchableInfo.Builder()
                .setSearchableComponent(new ComponentName(this, SearchActivity.class))
                .setSearchableText(entryText)
                .build());
        searchView.setSearchCallback(new SearchView.OnSearchClickListener() {
            @Override
            public void onSearchClick() {
                onSearchRequested();
            }
        });
    }

    @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));
    }
}
```
Searching with SearchManager

```java
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:

```java
@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 (ListAdapter) adapter;
}
```

The logic in the searchItems() method that actually finds the matches looks like:

```java
List<String> results = new ArrayList<String>();
for (String item : items) {
    if (item.indexOf(query) > -1) {
        results.add(item);
    }
}
return results;
```
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:

```xml
<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"/>
        <meta-data
          android:name="android.app.searchable"
          android:resource="@xml/searchable"/>
      </intent-filter>
    </activity>
  </application>
</manifest>
```
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.
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.

![Image of 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:
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:
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.
Below, we have a LoremSuggestionProvider, extending SearchRecentSuggestionsProvider, that also supplies a “bridge” for the search activity to record searches:

```java
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()`.
That SearchRecentSuggestions is then used by our LoremSearch class, inside its searchItems() method that actually examines the list of nonsense words for matches:

```java
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.

The result is that when we do our local search, we get the drop-down of past searches as suggestions:

![Android local search popup, showing the IME and a previous search](image)

*Figure 509: 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).

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:

![Searchable Items settings screen](image)

*Figure 510: 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:
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:
Even here, we do not see the actual suggestion. However, if the user clicks on that item, your suggestions then take over the list:
Again, Android is not showing actual data from your application – our list of nonsense 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.
Handling System Events

If you have ever looked at the list of available `Intent` actions in the SDK documentation for the `Intent` class, you will see that there are lots of possible actions.

There are even actions that are not listed in that spot in the documentation, but are scattered throughout the rest of the SDK documentation.

The vast majority of these you will never raise yourself. Instead, they are broadcast by Android, to signify certain system events that have occurred and that you might want to take note of, if they affect the operation of your application.

This chapter examines a couple of these, to give you the sense of what is possible and how to make use of these sorts of events. Note that we examined another similar one of these, to get control at boot time, back in the chapter on `AlarmManager`.

Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on `BroadcastReceiver`. Also, it might be a good idea to read the section on the `BOOT_COMPLETED` system broadcast in the chapter on `AlarmManager`.

I Sense a Connection Between Us…

Generally speaking, Android applications do not care what sort of Internet connection is being used — 3G, GPRS, WiFi, `lots of trained carrier pigeons`, or whatever. So long as there is an Internet connection, the application is happy.
Sometimes, though, you may specifically want WiFi. This would be true if your application is bandwidth-intensive and you want to ensure that, should WiFi stop being available, you cut back on your work so as not to consume too much 3G/GPRS bandwidth, which is usually subject to some sort of cap or metering.

There is an `android.net.wifi.WIFI_STATE_CHANGED` Intent that will be broadcast, as the name suggests, whenever the state of the WiFi connection changes. You can arrange to receive this broadcast and take appropriate steps within your application.

This Intent requires no special permission. Hence, all you need to do is register a `BroadcastReceiver` for `android.net.wifi.WIFI_STATE_CHANGED`, either via `registerReceiver()`, or via the `<receiver>` element in `AndroidManifest.xml`, such as the one shown below, from the `SystemEvents/OnWiFiChange` sample project:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest
    android:versionCode="1"
    android:versionName="1.0"
    package="com.commonsware.android.sysevents.wifi"
    xmlns:android="http://schemas.android.com/apk/res/android">
    <application
        android:icon="@drawable/cw"
        android:label="@string/app_name">
        <receiver android:name=".OnWiFiChangeReceiver">
            <intent-filter>
                <action android:name="android.net.wifi.WIFI_STATE_CHANGED" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

All we do in the manifest is tell Android to create an `OnWiFiChangeReceiver` object when a `android.net.wifi.WIFI_STATE_CHANGED` Intent is broadcast, so the receiver can do something useful.

In the case of `OnWiFiChangeReceiver`, it examines the value of the `EXTRA_WIFI_STATE` “extra” in the supplied Intent and logs an appropriate message:

```java
package com.commonsware.android.sysevents.wifi;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.net.wifi.WifiManager;
import android.util.Log;

public class OnWiFiChangeReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        int wifiState = intent.getIntExtra("android.net.wifi.EXTRA_WIFI_STATE", 0);
        switch (wifiState) {
            case WifiManager.WIFI_STATE_ENDED:
                Log.i("WiFi", "WiFi stopped available");
                break;
            case WifiManager.WIFI_STATE_AWAY:
                Log.i("WiFi", "WiFi connection lost");
                break;
            case WifiManager.WIFI_STATE_AWAY_SUSPENDED:
                Log.i("WiFi", "WiFi connection lost (suspended)");
                break;
            case WifiManager.WIFI_STATE_NO_PS:
                Log.i("WiFi", "WiFi network cannot support power saving");
                break;
            case WifiManager.WIFI_STATE_NO_PEERS:
                Log.i("WiFi", "WiFi network is not peer-to-peer");
                break;
            case WifiManager.WIFI_STATE_NO_SCAN:
                Log.i("WiFi", "WiFi scan is not available");
                break;
            case WifiManager.WIFI_STATE_NO_AVAILABLE:
                Log.i("WiFi", "WiFi not available or failed to find"
```
public void onReceive(Context context, Intent intent) {
    int state = intent.getIntExtra(WifiManager.EXTRA_WIFI_STATE, -1);
    String msg = null;

    switch (state) {
        case WifiManager.WIFI_STATE_DISABLED:
            msg = "is disabled";
            break;

        case WifiManager.WIFI_STATE_DISABLING:
            msg = "is disabling";
            break;

        case WifiManager.WIFI_STATE_ENABLED:
            msg = "is enabled";
            break;

        case WifiManager.WIFI_STATE_ENABLING:
            msg = "is enabling";
            break;

        case WifiManager.WIFI_STATE_UNKNOWN:
            msg = "has an error";
            break;

        default:
            msg = "is acting strangely";
            break;
    }

    if (msg != null) {
        Log.d("OnWiFiChanged", "WiFi " + msg);
    }
}

The EXTRA_WIFI_STATE “extra” tells you what the state has become (e.g., we are now disabling or are now disabled), so you can take appropriate steps in your application.

Note that, to test this, you will need an actual Android device, as the emulator does not specifically support simulating WiFi connections.

**Feeling Drained**

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. In the preceding section, we saw how you could find out when WiFi was disabled, so you might not use as much bandwidth when on 3G/GPRS. However, not every application uses so much bandwidth as to make this optimization worthwhile.
However, most applications are impacted by battery life. Dead batteries run no apps.

So whether you are implementing a battery monitor or simply want to discontinue background operations when the battery gets low, you may wish to find out how the battery is doing.

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.

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 as shown in the preceding two sections.

In the SystemEvents/OnBattery sample project, you will find a layout containing a ProgressBar, a TextView, and an ImageView, to serve as a battery monitor:

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ProgressBar android:id="@+id/bar"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
    <LinearLayout android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <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 BatteryMonitor activity, which registers to receive the 
ACTION_BATTERY_CHANGED Intent in onResume() and unregisters in onPause():

```java
package com.commonsware.android.sysevents.battery;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.os.BatteryManager;
import android.widget.ProgressBar;
import android.widget.ImageView;
import android.widget.TextView;

public class BatteryMonitor extends Activity {
    private ProgressBar bar=null;
    private ImageView status=null;
    private TextView level=null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        bar=(ProgressBar)findViewById(R.id.bar);
        status=(ImageView)findViewById(R.id.status);
        level=(TextView)findViewById(R.id.level);
    }

    @Override
    public void onResume() {
        super.onResume();

        registerReceiver(onBatteryChanged,
                         new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
    }

    @Override
    public void onPause() {
        super.onPause();

        unregisterReceiver(onBatteryChanged);
    }

    BroadcastReceiver onBatteryChanged=new BroadcastReceiver() {
        public void onReceive(Context context, Intent intent) {
            int pct=100*intent.getIntExtra("level", 1)/intent.getIntExtra("scale", 1);
            bar.setProgress(pct);
            level.setText(String.valueOf(pct));
        }
    };
```
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:

1. EXTRA_HEALTH, which should generally be BATTERY_HEALTH_GOOD
2. EXTRA_LEVEL, which is the proportion of battery life remaining as an integer, specified on the scale described by the scale “extra”
3. EXTRA_PLUGGED, which will indicate if the device is plugged into AC power (BATTERY_PLUGGED_AC) or USB power (BATTERY_PLUGGED_USB)
4. EXTRA_SCALE, which indicates the maximum possible value of level (e.g., 100, indicating that level is a percentage of charge remaining)
5. EXTRA_STATUS, which will tell you if the battery is charging (BATTERY_STATUS_CHARGING), full (BATTERY_STATUS_FULL), or discharging (BATTERY_STATUS_DISCHARGING)
6. EXTRA_TECHNOLOGY, which indicates what sort of battery is installed (e.g., "Li-Ion")
7. EXTRA_TEMPERATURE, which tells you how warm the battery is, in tenths of a degree Celsius (e.g., 213 is 21.3 degrees Celsius)
8. EXTRA_VOLTAGE, indicating the current voltage being delivered by the battery, in millivolts
**Handling System Events**

In the case of BatteryMonitor, when we receive an ACTION_BATTERY_CHANGED Intent, we do three things:

- We compute the percentage of battery life remaining, by dividing the level by the scale
- We update the ProgressBar and TextView to display the battery life as a percentage
- 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:

![Battery Monitor screenshot](image)

*Figure 514: The BatteryMonitor application*

**Sticky Intents and the Battery**

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.

**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.

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).
Other Power Triggers

If you are only interested in knowing when the device has been attached to, or detached from, a source of external power, there are different broadcast Intent actions you can monitor: ACTION_POWER_CONNECTED and ACTION_POWER_DISCONNECTED. These are only broadcast when the power source changes, not just every time the battery changes charge level. Hence, these will be more efficient, as your code will be invoked less frequently. Better still, you can use manifest-registered broadcast receivers for these, bypassing the limits the system puts on ACTION_BATTERY_CHANGED.
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*
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 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 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**

Our sample revolves around a scripting language called [BeanShell](https://commonsware.com). BeanShell is, in effect, a Java interpreter for Java. We will go into greater detail about BeanShell elsewhere in this book. For here, most of what you need to know is that you can have BeanShell interpret a chunk of source code by creating an `Interpreter` object and calling `eval()`.

In the `AdvServices/Binding` sample project, we have an activity, displaying a fragment, containing the world's smallest IDE:

![Service Binding Demo](image)

*Figure 515: Binding Demo, As Initially Launched*
When the user types in some Java code and clicks the button, we want to execute that code. And, in this case, we will use a service and the binding pattern to do so.

We start by defining an interface that will serve as the “contract” between the client (fragment) and service. This interface, IScript, contains a single executeScript() method:

```java
package com.commonsware.android.advservice.binding;

// Declare the interface.
interface IScript {
    void executeScript(String script);
}
```

Our service, BshService, implements just one method, onBind(), which returns an instance of a BshBinder:

```java
package com.commonsware.android.advservice.binding;

import android.app.Service;
import android.content.Context;
import android.content.Intent;
import android.os.Binder;
import android.os.IBinder;
import android.util.Log;
import bsh.EvalError;
import bsh.Interpreter;

public class BshService extends Service {
    @Override
    public IBinder onBind(Intent intent) {
        return (new BshBinder(this));
    }

    private static class BshBinder extends Binder implements IScript {
        private Interpreter i = new Interpreter();

        BshBinder(Context ctxt) {
            try {
                i.set("context", ctxt);
            }
            catch (EvalError e) {
                Log.e("BshService", "Error executing script", e);
            }
        }

        public void executeScript(String script) {
            try {
                i.eval(script);
            }
            catch (bsh.EvalError e) {
```
**REMOTE SERVICES AND THE BINDING PATTERN**

```java
Log.e("BshService", "Error executing script", e);
}
};
}
```

BshBinder implements the IScript interface and is where our BeanShell “business logic” resides:

- In the BshBinder initializers, we create an instance of the BeanShell Interpreter class
- In the BshBinder constructor, we inject an object — the BshService instance — into the BeanShell interpreted environment as what amounts to a global object, named context
- In the executeScript() method, we just pass the supplied BeanShell source to the eval() method of our Interpreter

Our fragment, BshFragment, loads our layout, res/layout/main.xml, containing a Button and a multi-line EditText:

```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">
    <Button
        android:id="@+id/eval"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/go"/>
    <EditText
        android:id="@+id/script"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="top"
        android:inputType="textMultiLine"/>
</LinearLayout>
```

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:

```java
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
```
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 execute a
BeanShell script.

In onActivityCreated() of our fragment, we bind to the service:

```java
@Override
public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);

    getActivity().getApplicationContext()
        .bindService(new Intent(getActivity(), BshService.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.

Some time after onActivityCreated() 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
IScript interface to the service, and we can enable the Button:

```java
@Override
public void onServiceConnected(ComponentName className, IBinder binder) {
    service=(IScript)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 IScript object and disables our Button:

```java
@Override
public void onServiceDisconnected(ComponentName className) {
    disconnect();
}

private void disconnect() {
    service=null;
    btn.setEnabled(false);
}
```

And, when our fragment is destroyed, we unbind from the service (using the same Context as before, from `getApplicationContext()`) and `disconnect()`:

```java
@Override
public void onDestroy() {
    get.Activity().getApplicationContext().unbindService(this);
    disconnect();
    super.onDestroy();
}
```

However, in between `onServiceConnected()` and either `onServiceDisconnected()` or `onDestroy()`, the user can type in and submit a script, triggering a call to `onClick()` when the user clicks the “Go!” button:

```java
@Override
public void onClick(View view) {
    EditText script=(EditText) getView().findViewById(R.id.script);
    String src=script.getText().toString();
    service.executeScript(src);
}
```

Here, we get the source code from the EditText and pass it to the IScript interface for processing. In this case, we happen to do so on the main application thread, which means that the script will be evaluated on the main application thread as well.

The result is that the user can enter in a script — including referencing our context global Context object — and execute it by clicking “Go!”:
For example, you can type in:

```java
import android.widget.Toast;
Toast.makeText(context, "Hi, Mom!", Toast.LENGTH_LONG).show();
```

and it will display your requested Toast:
The BshServiceDemo activity that uses our BshFragment not only adds BshFragment via a FragmentTransaction, but it offers an overflow menu item to launch another instance of the activity itself:

```java
class BshServiceDemo extends SherlockFragmentActivity {
    @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 BshFragment()).commit();
        }
    }
}
```

**Figure 517: Binding Demo, Running a Script**
If you run multiple instances of the activity — and hence multiple instances of the fragment — you will see that each gets its own binding to the service, and each can execute scripts with the help of that service.

### 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 COM, 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. For example, here is some AIDL:

```
package com.commonsware.android.advservice;

// Declare the interface.
interface IScript {
    void executeScript(String script);
}
```

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 IScript { ... }). 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 interface, which is Android's flavor of serialization

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; we included in for the AIDL for enable() just for illustration purposes.
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.

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., IScriptStub)
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 AdvServices/RemoteService and AdvServices/RemoteClient sample projects — integrate BeanShell into a remote service, along the lines of the local binding sample from earlier in this chapter. If you actually wanted to use scripting in an Android application, with scripts loaded off of the Internet, isolating their execution into a service might not be a bad idea. In the
service, those scripts are sandboxed, only able to access files and APIs available to that service. The scripts cannot access your own application’s databases, for example. If the script-executing service is kept tightly controlled, it minimizes the mischief a rogue script could possibly do.

**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.

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
<?xml version="1.0" encoding="utf-8"?>
<manifest
    android:versionCode="1"
    android:versionName="1.0"
    package="com.commonsware.android.advservice"
    xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-sdk
        android:minSdkVersion="3"
        android:targetSdkVersion="6" />
    <supports-screens
        android:largeScreens="false"
        android:normalScreens="true"
        android:smallScreens="false" />
    <application
        android:icon="@drawable/cw"
        android:label="@string/app_name">
        <service
            android:name=".BshService">
            <intent-filter>
                <action
                    android:name="com.commonsware.android.advservice.IScript" />
            </intent-filter>
        </service>
    </application>
</manifest>
```

Here, we say that the service can be identified by the name `com.commonsware.android.advservice.IScript`. 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.

---

1995

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

Beyond the manifest, the service implementation is not too unusual. There is the AIDL interface, IScript:

```java
package com.commonsware.android.advservice;

// Declare the interface.
interface IScript {
    void executeScript(String script);
}
```

And there is the actual service class itself, BshService:

```java
package com.commonsware.android.advservice;

import android.app.Service;
import android.content.Context;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
import bsh.EvalError;
import bsh.Interpreter;

public class BshService extends Service {
    @Override
    public IBinder onBind(Intent intent) {
        return (new BshBinder(this));
    }

    private static class BshBinder extends IScript.Stub {
        private Interpreter i=
            new Interpreter();

        BshBinder(Context ctxt) {
            try {
                i.set("context", ctxt);
            }
            catch (EvalError e) {
                Log.e("BshService", "Error executing script", e);
            }
        }

        @Override
        public void executeScript(String script) {
            try {
                i.eval(script);
            }
            catch (bsh.EvalError e) {
                Log.e("BshService", "Error executing script", e);
            }
        }
    }
}
```
This is identical to the local binding example, with one key difference: BshBinder now extends IScriptStub rather than the generic Binder class.

Also note that, in this implementation, the script is executed directly by the service on the calling thread. One might think this is not a problem, since the service is in its own process and, therefore, cannot possibly be using the client’s UI thread. However, AIDL IPC calls are synchronous, so the client will still block waiting for the script to be executed. This too will be corrected later in this chapter.

The Client

The client — a revised version of BshFragment — connects to the remote service to ask it to execute BeanShell scripts on the user’s behalf:

```java
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import com.commonsware.android.advservice.IScript;

public class BshFragment extends Fragment implements OnClickListener, ServiceConnection {
    private IScript service=null;
    private Button btn=null;

    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        View result=inflater.inflate(R.layout.main, container, false);
        btn=(Button)result.findViewById(R.id.eval);
        btn.setOnClickListener(this);
        btn.setEnabled((service!=null));

        return(result);
    }

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        setRetainInstance(true);
        get.Activity().getApplicationContext().bindService(new Intent("com.commonsware.android.advservice.IScript"),
                        this, Context.BIND_AUTO_CREATE);
    }
}
```
REMOTE SERVICES AND THE BINDING PATTERN

```java
@Override
public void onDestroy() {
    getActivity().getApplicationContext().unbindService(this);

    super.onDestroy();
}

@Override
public void onClick(View view) {
    EditText script = (EditText) getView().findViewById(R.id.script);
    String src = script.getText().toString();

    try {
        service.executeScript(src);
    } catch (RemoteException e) {
        Toast.makeText(getActivity(), e.toString(), Toast.LENGTH_LONG).show();
    }
}

@Override
public void onServiceConnected(ComponentName className, IBinder binder) {
    service = IScript.Stub.asInterface(binder);
    btn.setEnabled(true);
}

@Override
public void onServiceDisconnected(ComponentName className) {
    service = null;
}
```

This is the same as with the local binding scenario, except:

- We use a different Intent with `bindService()`, one identifying the remote service by name
- Our `onServiceConnected()` uses `IScript.Stub.asInterface()` to convert the raw `IBinder` into an `IScript` object for use
- We have to catch a `RemoteException` when we try to call `executeScript()`, in case the service crashed or is otherwise inaccessible at this moment

Note that the client needs its own copy of `IScript.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 `IScript` 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 enter in Beanshell code and have it be executed by the service. Note, though, that you cannot perform UI operations (e.g., raise a Toast) from the service, because in that process, the executeScript() method is not running on the main application thread. If you choose some script that is long-running, you will see that the “Go!” button is blocked until the script is complete.

**Servicing the Service**

The preceding section outlined two flaws in the implementation of the Beanshell remote service:

- The client received no results from the script execution
- The client blocked waiting for the script to complete

If we were not worried about the blocking-call issue, we could simply have the executeScript() exported API return some sort of result (e.g., toString() on the result of the Beanshell eval() call). However, that would not solve the fact that calls to service APIs are synchronous even for remote services.

Another approach would be to pass some sort of callback object with executeScript(), 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.

Fortunately, this is eminently doable, as you will see in this section, and the accompanying samples (AdvServices/RemoteServiceEx and AdvServices/RemoteClientEx).

**Callbacks via AIDL**

AIDL does not have any concept of direction. It just knows interfaces 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 and clients accessing — 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 IScriptResult.aidl file:

```java
package com.commonsware.android.advservice;

// Declare the interface.
interface IScriptResult {
    void success(String result);
    void failure(String error);
}
```

Then, we can augment IScript itself, to pass an IScriptResult with `executeScript()`:

```java
package com.commonsware.android.advservice;

import com.commonsware.android.advservice.IScriptResult;

// Declare the interface.
interface IScript {
    void executeScript(String script, IScriptResult cb);
}
```

Notice that we need to specifically import IScriptResult, 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.

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 IScriptResult. On `success()`, we can do something like raise a `Toast`; on `failure()`, we can perhaps show an `AlertDialog`.

The catch is that we cannot be certain we are being called on the UI thread in our callback object.

So, the safest way to do that is to make the callback object use something like `runOnUiThread()` to ensure the results are displayed on the UI thread. And, of
course, we need to update our call to `executeScript()` to pass the callback object to the remote service.

```java
package com.commonsware.android.advservice.client;

import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.os.RemoteException;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import com.commonsware.android.advservice.IScript;
import com.commonsware.android.advservice.IScriptResult;

public class BshFragment extends Fragment implements OnClickListener,
        ServiceConnection {
    private IScript service=null;
    private Button btn=null;

    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        View result=inflater.inflate(R.layout.main, container, false);
        btn=(Button)result.findViewById(R.id.eval);
        btn.setOnClickListener(this);
        btn.setEnabled((service!=null));

        return(result);
    }

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        setRetainInstance(true);
        get.Activity().getApplication().
                bindService(new Intent("com.commonsware.android.advservice.IScript"),
                        this, Context.BIND_AUTO_CREATE);
    }
}
```
public void onDestroy() {
    getActivity().getApplicationContext().unbindService(this);

    super.onDestroy();
}

@Override
public void onClick(View view) {
    EditText script=(EditText)getView().findViewById(R.id.script);
    String src=script.getText().toString();

    try {
        service.executeScript(src, callback);
    } catch (RemoteException e) {
        Toast.makeText(getActivity(), e.toString(), Toast.LENGTH_LONG).show();
    }
}

@Override
public void onServiceConnected(ComponentName className, IBinder binder) {
    service=IScript.Stub.asInterface(binder);
    btn.setEnabled(true);
}

@Override
public void onServiceDisconnected(ComponentName className) {
    service=null;
}

private final IScriptResult.Stub callback=new IScriptResult.Stub() {
    public void success(final String result) {
        getActivity().runOnUiThread(new Runnable() {
            public void run() {
                Toast.makeText(getActivity(), result, Toast.LENGTH_LONG).show();
            }
        });
    }

    public void failure(final String error) {
        getActivity().runOnUiThread(new Runnable() {
            public void run() {
                Toast.makeText(getActivity(), error, Toast.LENGTH_LONG).show();
            }
        });
    }
};
Revising the Service

The service also needs changing, to both execute the scripts asynchronously and use the supplied callback object for the end results of the script’s execution.

BshService from AdvServices/RemoteServiceEx uses a ThreadPoolExecutor to manage a background thread. An ExecuteScriptJob wraps up the script and callback; when the job is eventually processed, it uses the callback to supply the results of the eval() (on success) or the message of the Exception (on failure):

```java
package com.commonsware.android.advservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
import bsh.Interpreter;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class BshService extends Service {
    private final ExecutorService executor =
            new ThreadPoolExecutor(1, 1, 60, TimeUnit.SECONDS,
                        new LinkedBlockingQueue<Runnable>());
    private final Interpreter i = new Interpreter();

    @Override
    public void onCreate() {
        super.onCreate();

        try {
            i.set("context", this);
        } catch (bsh.EvalError e) {
            Log.e("BshService", "Error executing script", e);
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return new BshBinder();
    }

    @Override
    public void onDestroy() {
        executor.shutdown();

        super.onDestroy();
    }
}
```
private class ExecuteScriptJob implements Runnable {
  IScriptResult cb;
  String script;

  ExecuteScriptJob(String script, IScriptResult cb) {
    this.script=script;
    this.cb=cb;
  }

  @Override
  public void run() {
    try {
      cb.success(i.eval(script).toString());
    } catch (Throwable e) {
      Log.e("BshService", "Error executing script", e);
      try {
        cb.failure(e.getMessage());
      } catch (Throwable t) {
        Log.e("BshService", "Error returning exception to client", t);
      }
    }
  }
}

private class BshBinder extends IScript.Stub {
  @Override
  public void executeScript(String script, IScriptResult cb) {
    executor.execute(new ExecuteScriptJob(script, cb));
  }
};

Notice that the service’s own API just needs the IScriptResult 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.

The net result is that the client can call the service and get its results without tying up the client’s UI thread.

You may be wondering why we do not simply use an AsyncTask. The reason is that remote service methods exposed by AIDL are not invoked on the main application thread — one of the few places in Android where Android calls your code from a background thread. An AsyncTask expects to be created on the main application thread.
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 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.
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.
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:
ADVANCED MANIFEST TIPS

- 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
<?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"/>
</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., target=android-8 in project.properties for those of you building via Ant, 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.
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.
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 518: Warning when unmounting the SD card](image-url)
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:

![Manage Applications list, with an application shown from a removed SD card](image)

*Figure 519: The Manage Applications list, with an application shown from a removed SD card*
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.
**ADVANCED MANIFEST TIPS**

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
<?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>
</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.
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](https://play.google.com/store/apps/details?id=com.commonsware.android.alias) is a [lockscreen app widget](https://developer.android.com/guide/topics/ui/app-capabilities) 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
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:

```xml
<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 metadata, an app can ask for PackageManager.GET_META_DATA as a flag on PackageManager methods for introspection, like queryIntentActivities().

---

2017

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:

```java
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:

```java
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.
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.
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:
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:
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().
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 520: The home screen customization options list](image)

Choose Shortcuts, and scroll down to find QuickSender in the list:
Click the QuickSender entry, which will bring up our activity with the form to define what to send:
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:
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:
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.

### ShareActionProvider

Earlier in this book, we saw how you can bring up a chooser when using `startActivity()` on an implicit Intent action, such as ACTION_SEND.

And, earlier in this book, we saw how PackageManager and methods like `queryIntentActivities()` can be used to create your own means for the user to choose some implementation of an implicit Intent action, such as how a home screen lets the user choose an implementation of ACTION_MAIN.
Nowadays, there is another option, if you are using the action bar (native or via ActionBarSherlock): 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/Share sample project.

Our activity — MainActivity — will utilize the action bar, in this case via ActionBarSherlock. Its action bar items are contained in a res/menu/actions.xml file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
      android:id="@+id/share"
      android:actionProviderClass="com.actionbarsherlock.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, specifically the one from ActionBarSherlock. There is an equivalent class for the native action bar in the Android SDK, should you be developing without ActionBarSherlock.

Our activity UI is simply a large EditText widget:

```xml
          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:

```java
package com.commonsware.android.sap;

import android.content.Intent;
```

2029
public class MainActivity extends SherlockActivity implements ShareActionProvider.OnShareTargetSelectedListener, TextWatcher {
    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
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.
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.

AndroidViews.net is probably the leading contender at present, though their emphasis is definitely on UI components. OpenIntents was the original catalog, but it does not get a lot of attention nowadays.
Beyond that, look for recommendations in StackOverflow answers, blog posts, and the like.

**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.

Most published Android library projects include full source code, as they are usually open source projects. However, it is possible to create a binary-only Android library project, one where the source code is replaced by a JAR that accompanies the resources. Google’s Play Services SDK is distributed as one such “binary-only” library project.

As was covered earlier in the book, using an Android library project involves making it available to your build environment (e.g., importing it into your Eclipse workspace) and then teaching your application project to reference the library project (e.g., Project > Properties > Android in Eclipse). This approach works regardless of whether the library project includes source code or not.
**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.

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
  - 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.
ReUsable Components

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 a custom Ant target that uses the standard Ant `<jar>` task to package things up:

```
<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>
```

This way, running `ant jar` creates a JAR. The three `<exclude>` elements strip out compiled classes from your project’s `gen/` directory that you would not want to ship — BuildConfig and everything from `R.java`.

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.

**Binary-Only Library Projects**

It is possible to create a binary-only library project, one where your source code is replaced by a JAR. This can be useful for proprietary library 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 complex scenarios like that, you will need to hold out for the AAR solution that is forthcoming.

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, given an Ant jar target as shown earlier in this chapter, an Ant target to create a distribution ZIP might be:

```xml
<target name="dist" depends="jar">
  <copy todir="/tmp/WhateverYouWantToCallYourLibrary/libs">
    <fileset dir="libs/"/>
  </copy>
  <copy todir="/tmp/WhateverYouWantToCallYourLibrary/res">
    <fileset dir="res/"/>
  </copy>
  <copy>
    <fileset dir="/tmp/WhateverYouWantToCallYourLibrary/src">
      <fileset dir="libs/"/>
      <fileset dir="res/"/>
      <fileset dir="/tmp/WhateverYouWantToCallYourLibrary"/>
    </fileset>
  </copy>
  <zip destfile="/tmp/WhateverYouWantToCallYourLibrary.zip">
    <fileset dir="/tmp/WhateverYouWantToCallYourLibrary/libs"/>
    <fileset dir="/tmp/WhateverYouWantToCallYourLibrary/res"/>
    <fileset dir="/tmp/WhateverYouWantToCallYourLibrary/src"/>
    <fileset dir="/tmp/WhateverYouWantToCallYourLibrary"/>
    <zip destfile="/tmp/WhateverYouWantToCallYourLibrary.zip">
      <fileset dir="/tmp/WhateverYouWantToCallYourLibrary/libs"/>
      <fileset dir="/tmp/WhateverYouWantToCallYourLibrary/res"/>
      <fileset dir="/tmp/WhateverYouWantToCallYourLibrary/src"/>
      <fileset dir="/tmp/WhateverYouWantToCallYourLibrary"/>
    </zip>
  </zip>
</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.

### The Future: AAR

Google is hard at work on creating a new build system, to replace the combination of Ant and Eclipse’s internal build system. This new build system, based upon [Gradle](https://gradle.org), will package Android library projects as AAR (.aar) files. These files, in principle, should package up the entire library project, in binary form, as a single file.

### 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.

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 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
Trail: Scripting Languages
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.
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 there 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.
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.
THE ROLE OF SCRIPTING LANGUAGES

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.
THE ROLE OF SCRIPTING LANGUAGES

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
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.
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:
Then, to install additional interpreters, choose Add from the option menu. You will be given a roster of SL4A-compatible interpreters to choose from:
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:
Click the Install button, and SL4A will download and install the interpreter's component parts:
Figure 528: 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:
The Scripting Layer for Android

![Image](image.jpg)

*Figure 529: 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 $\text{SL4A}/$ 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:
Figure 530: The list of SL4A scripts

Tapping on any of these scripts will bring up a “quick actions” balloon:
Click the little shell icon to run it, showing its terminal output along the way:
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.
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 533: The add-script language selection dialog](image)

(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:
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.
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:

```
>>>import android
>>>droid=android.Android()
>>>droid.speak(time.strftime("%I:%M %p on %A, %B %d, %Y "))
```

Figure 535: 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.
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:

![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._
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:
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:
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](https://commonsware.com) in SL4A for packaging scripts as APK files for wider distribution.
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.
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.
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.
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
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
JVM Scripting Languages

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
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
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:

```java
package com.commonsware.abj.interp;
import android.os.Bundle;
public interface IInterpreter {
  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:

```java
package com.commonsware.abj.interp;
import android.os.Bundle;

public class EchoInterpreter implements IInterpreter {
  public Bundle executeScript(Bundle input) {
    return(input);
  }
}
```

A somewhat more elaborate sample is the `SQLLiteInterpreter`:

```java
package com.commonsware.abj.interp;
import android.database.Cursor;
```
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.
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:

```java
@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);
            }
        }
    }
}
```
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:

```java
private void success(Intent intent, Bundle result) {
    Intent data = new Intent();
    data.putExtras(result);
    data.putExtra(ResultCode, SUCCESS);
}
```
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:

```java
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);
        }
    }
}
```
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 IInterpreter 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:

```xml
<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 IInterpreter 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.
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:

```java
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:

```java
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'
  }
```

2080
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.
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:

```java
default 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);
            } catch (InterpreterException e) {

```
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.
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 IInterpreter facade on Rhino is incrementally more difficult than it is for BeanShell, but not by that much:

```java
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);
      }
      finally {
        ctxt=null;
      }
    }

    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
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.
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.
JUnit and Android

Presumably, you will want to test your code, beyond just playing around with it yourself by hand.

To that end, Android includes the JUnit test framework in the SDK, along with special test classes that will help you build test cases that exercise Android components, like activities and services. Even better, Android has “gone the extra mile” and can pre-generate your test harness for you, to make it easier for you to add in your own tests.

**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.

**You Get What They Give You**

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 set up to test various facets of one of our “show a list of 25 nonsense words” samples.
To create one of these test projects, you can either use Eclipse or the command line, as is the case with regular Android projects.

**Eclipse**

In 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:

![Eclipse Android Test Project Wizard, First Page](image)

The second page of the wizard has you pick from one of your Android projects the one that you wish to test:
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:
Command Line

From the command line, you use `android create project` to create a regular Android project. To create a project designed to test another project — what we will call a “test project” — you use the `android create test-project` command. From Eclipse, you can create a test project using the appropriate wizard. You will need to tell it which project to test, where you want the test project to reside, etc.

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 preparatory 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:

```java
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() {
        assertTrue(1==1);
    }
}
```

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 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:

```java
package com.commonsware.android.abf.test;

import android.test.ActivityInstrumentationTestCase2;
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();

    ActionBarFragmentActivity activity = getActivity();
    list = (ListView) activity.findViewById(android.R.id.list);
}

public void testListCount() {
    assertTrue(list.getAdapter().getCount() == 25);
}

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. In testListCount(), we check our ListAdapter in the ListView to make sure we have all 25 of our nonsense words at the outset. This is fairly trivial and non-interactive. However, you could use methods like sendKeys() to simulate user input, to drive changes in your UI, so you can confirm the results worked as expected.

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. At runtime, the contents of both your regular application and the test application are combined into a single process in a single copy of the Dalvik VM, which is why your test code can access your application classes.

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 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:

```java
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.

**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:

```java
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.

Running Your Tests

As with most things in Android, you can either use Eclipse or the command line to run your test suites.

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”. 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:

![Figure 542: Eclipse “Run As” Dialog for JUnit Test](image)

The results will be displayed in a JUnit view added to your Eclipse workspace, showing the successful and failed tests:
JUnit and Android

![Eclipse Android JUnit Test Results](image)

**Figure 543: Eclipse Android JUnit Test Results**

**Command Line**

Android ships with a very rudimentary console JUnit runner, called `InstrumentationTestRunner`. Since this class resides in the Android environment (emulator or device), you need to invoke the runner to run your tests on the emulator or device itself. To do this, you can run `ant test` from a console. You should see results akin to:

```bash
  test:
    [echo] Running tests ...
    [exec]  com.commonsware.android.abf.test.DemoActivityTest:..
    [exec]  com.commonsware.android.abf.test.SillyTest:.
    [exec]  Test results for InstrumentationTestRunner=.....
    [exec]  Time: 0.173
    [exec]  [exec] OK (5 tests)
    [exec]
```
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.
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:

```python
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)
device.press('KEYCODE_DPAD_DOWN', MonkeyDevice.DOWN_AND_UP)
```
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:

```bash
adb shell monkey -p com.commonsware.android.abf -v --throttle 100 600
```

(substituting the package name of a project on your device or emulator for `com.commonsware.android.abf`)

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 com.commonsware.android.abf`).
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](https://commonsware.com/monkeystack/) in the SDK’s Developer’s Guide covers all of that and more.
Testing with UIAutomator

Yet another approach for testing Android applications arose with the R21 release of the Android developer tools, in the form of `uiautomator`. This blends JUnit and Test Monkey approaches, offering some unique advantages over either of those individual frameworks.

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.

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.

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.

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 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:

```
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
• 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:

```java
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().packageName("com.commonsware.android.abf"),
                                            "Action Bar Fragment Demo");
    ourApp.clickAndWaitForNewWindow();

    UiObject appValidation = 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:

```java
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 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:

```java
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.

```java
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.

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:

```java
UiScrollable appViews = new UiScrollable(new UiSelector().scrollable(true));
appViews.setAsHorizontalList();
```
Testing with UIAutomator

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:

```java
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:

```java
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.

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:

```java
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:

```java
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
- 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

```java
public void testContents() throws UiObjectNotFoundException {
    UiScrollable words =
        new UiScrollable(
            new UiSelector().className("android.widget.ListView"));

    words.setAsVerticalList();

    for (String s : items) {
```
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

```java
public void testAdd() throws UiObjectNotFoundException {
    UiObject add = new UiObject(new UiSelector().text("Word"));
    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:

```java
@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, particularly Eclipse integration.

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/
```

Executing uiautomator

To actually run the tests, you will use `adb shell` to run the `uiautomator runtest` 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 is 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.

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`.

**Limitations of uiautomator**

While handy, `uiautomator` is not without its issues. In addition to the ones mentioned throughout this chapter:

- There is no way for `uiautomator` to test multi-touch operations at this time.
- `uiautomator` works with ARM-powered devices and emulators. It notably does not work with x86 emulators, due to some differences between the actual Linux shell that is invoked via the `adb shell` command. This is
unfortunate, given the clear speed advantages of the x86 emulators. With luck, this will be repaired in a future edition of the Intel-supplied emulator images.
Trail: Gradle and the New Build System
Introducing Gradle

Two major shifts in Android application development began in mid–2013 before hitting the mainstream in 2014. One is Android Studio, the new IDEA-based IDE. The other is “the new build system”, which is based upon a build engine called Gradle.

This chapter, and the chapters that follow, will give you a foundation in the use of this new build system, as it will be a useful adjunct to developers using IDEs, as well as being a core capability for developers who have elected to not use any IDE.

This chapter specifically will explain what Gradle is, how it relates to Android application development, how to get Gradle on your development machine, and some security issues with Gradle-based development.

Prerequisites and Warnings

Understanding this chapter requires that you have read the core chapters of this book.

NOTE: This chapter discusses a pre-release technology. The technology is undergoing rapid revision and suffers from a dearth of documentation. More so than with other chapters in this book, you may encounter behavior that differs from what is documented. As the saying goes, “your mileage may vary”.

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?

Primarily, Gradle is a replacement for Ant for doing builds at the command line.

Ant and Gradle are build systems. They are software designed to build other software, much as make is a build system. 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.

Ant used XML as the way of providing project-specific instructions for indicating what to build. By means of `<import>` elements, a project’s build file (e.g., `build.xml`) could incorporate common instructions. And Ant could be extended through custom JARs that defined new elements that a build file could use.

Gradle, at its core, does the same basic thing. However, rather than using XML and elements, it uses a domain-specific language (DSL) built on top of Groovy to accomplish the same ends.

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 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?

“The new build system” for Android apps is based upon Gradle. It is expected that Ant support will become deprecated and, eventually, abandoned.

Why Are We Moving to Gradle?

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, there are apparently limits to that solution. Gradle offers much better support in this area, 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.
How Does Gradle Relate to Android Studio?

Android Studio uses the new Gradle-based build system as its native approach for building Android projects. While IDEA 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 (October 2013), Eclipse has no ability to use Gradle 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 expected that sometime in the future, Google will attempt to have Eclipse be able to use Gradle for builds. How successful this attempt will be remains to be seen.

Obtaining Gradle

As with any build system, to use it, you need the build system’s engine itself. There are a few ways of going about this, described in the following sections.

Direct Installation

What most developers will wind up doing is installing Gradle directly.

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. This is the “Gradle Wrapper”, which will automatically download and install Gradle to an undocumented location on your development machine, then use it (instead of any existing installed copy of Gradle) for the project build.

However, this implies that you trust the source of the gradlew script.

If you are in an enterprise setting, and you know exactly who created gradlew, and therefore trust it, feel free to use it. On the other hand, do not use a gradlew script obtained from semi-random sources on the Internet – install Gradle yourself directly from the Gradle Web site, so that you are sure that you know where your Gradle installation came from.

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 0.5.x generation of Gradle for Android required Gradle 1.8
- The 0.7.x generation of Gradle for Android required Gradle 1.9
- The 0.8.x generation of Gradle for Android requires Gradle 1.10
- The 0.9.x generation of Gradle for Android requires Gradle 1.10, but works with 1.11

If you are using the gradlew 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 gradlew 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

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.

Some Brief Words About Maven

Maven is a tool-slash-site-slash-philosophy used by many Java developers outside of Android. You can roughly divide Maven into two parts:

- A build system (using the `mvn` command)
- Maven Central, a repository of “artifacts”, such as libraries

Gradle does not use the Maven build system, but rather replaces it. However, Gradle can integrate with Maven Central, so you can obtain libraries and such from it as part of the build process.

Learning More About Gradle

The Gradle Web site hosts documentation, links to Gradle-specific books, and links to other Gradle educational resources.

As the new build system is still in a pre-release state, the 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.
Gradle and Legacy 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).

However, Gradle is capable of building projects in either directory layout. This chapter will review how to add Gradle support to an existing Android project, without having to change your directory structure. Along the way, we will begin examining what Gradle build files look like, so you can start to create your own from scratch.

Prerequisites and Warnings

Understanding this chapter requires that you have read the chapter that introduces Gradle.

NOTE: This chapter discusses a pre-release technology. The technology is undergoing rapid revision and suffers from a dearth of documentation. More so than with other chapters in this book, you may encounter behavior that differs from what is documented. As the saying goes, “your mileage may vary”.

“Legacy”?

Here, “legacy directory structure” means a project tree that looks a bit like this:

...
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**

The default name for an Ant build file is `build.xml`. The default name for a Gradle build file is `build.gradle`. Just as you need a `build.xml` file to be able to build your project with Ant, 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.
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.

**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:

![Eclipse Export Wizard, First Page](image)

*Figure 545: Eclipse Export Wizard, First Page*

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 546: Eclipse Export Wizard, Second Page](image)

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:
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:
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, akin to the one that we will examine [later in this chapter](#).
- 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.
- 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.
**Hand-Writing**

Of course, you are welcome to create your own `build.gradle` from scratch, or by copying an existing one from elsewhere and making modifications to it. For advanced features, this will be the typical approach.

When you first use your `build.gradle` file, Gradle will create the `.gradle/` directory for you.

**Running a Gradle Build**

With Ant, to run Ant tasks, you run `ant ...` at a command prompt, where the `...` is replaced by one or more task names.

Gradle works much the same way, except with a `gradle` command instead of `ant`. What is substantially different is the roster of tasks and the results of the build.

**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:

```plaintext
: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
assembleRelease - Assembles all Release builds
assembleTest - Assembles the Test build for the Debug build
build - Assembles and tests this project.
bUILDDEPendents - Assembles and tests this project and all projects that depend on it.
bUILDNEEded - Assembles and tests this project and all projects it depends on.
clean - Deletes the build directory.
Build Setup tasks
```

---

2131
setupBuild - Initializes a new Gradle build. [incubating]

wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
dependencies - Displays all dependencies declared in root project 'Hello'.
dependencyInsight - Displays the insight into a specific dependency in root project 'Hello'.
help - Displays a help message
projects - Displays the sub-projects of root project 'Hello'.
properties - Displays the properties of root project 'Hello'.
tasks - Displays the tasks runnable from root project 'Hello'.

Install tasks
-------------
installDebug - Installs the Debug build
installTest - Installs the Test build for the Debug build
uninstallAll - Uninstall all applications.
uninstallDebug - Uninstalls the Debug build
uninstallRelease - Uninstalls the Release build
uninstallTest - Uninstalls the Test build for the Debug build

Verification tasks
------------------
check - Runs all checks.
connectedCheck - Runs all device checks on currently connected devices.
connectedInstrumentTest - Installs and runs the tests for Build 'Debug' on connected devices.
deviceCheck - Runs all device checks using Device Providers and Test Servers.
lint - Runs lint on all variants.

Rules
-----
Pattern: build<ConfigurationName>: Assembles the artifacts of a configuration.
Pattern: upload<ConfigurationName>: Assembles and uploads the artifacts belonging to a configuration.
Pattern: clean<TaskName>: Cleans the output files of a task.

To see all tasks and more detail, run with --all.

BUILD SUCCESSFUL
Total time: 7.341 secs

This list is dynamically generated based on the contents of build.gradle, notably including tasks defined by the android 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.
Gradle and Legacy Projects

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.

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, 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 Ant, the build output went in a bin/ directory. With Gradle, the build output goes into a build/ directory.

Specifically, your APKs will go into build/apk, with different APK editions based upon whether you did a debug or release build.

Gradle has a clean task that wipes out the build/ directory. However, note that Eclipse and Ant use the bin/ directory, so a Gradle clean will not “clean” the pre-compiled classes and such that Eclipse and Ant use. The reverse is also true: ant clean does not touch the build/ directory and so will not “clean” the Gradle pre-compiled classes.

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:
In this section, we will examine what is in this generated file and what it means to you, an Android app developer.
**buildscript**

The *buildscript* block in Gradle is where you configure the JARs and such that Gradle itself will use for interpreting the rest of the file. Hence, here you are not configuring your *project* so much as you are configuring the *build itself*.

The generated *build.gradle* file has code to bootstrap installation of Android SDK components necessary to complete the build:

```groovy
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.9.0'
    }
}
```

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

The *dependencies* block indicates what is required to be able to run the rest of the build script. *classpath 'com.android.tools.build:gradle:0.9.0'* is not especially well-documented by the Gradle team. However the *'com.android.tools.build:gradle:0.9.0'* 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 *0.9.0*

You will sometimes find these dependencies with a + instead of a specific version number in the last segment (e.g., 0.9.+). This means “0.9.0 or anything higher in the 0.9.x series”. This assumes that all of the 0.9.x series are compatible, meaning that a script for version 0.9.0 of the Gradle for Android plugin will also work with 0.9.1, etc.

The first time you run your build, with the *buildscript* block as shown above, Gradle will notice that you do not have this dependency. It will then visit [Maven Central](https://mvnrepository.com) and find the *com.android.tools.build* group and the *gradle* artifact within that group, then download version 0.9.0.
apply plugin

The 'com.android.tools.build:gradle:0.9.0' dependency contains an implementation of an android plugin for Gradle, to teach Gradle Android-specific constructs, such as those found in the android block covered later in this chapter. However, just downloading the dependency does not actually apply anything in it.

The apply plugin: 'android' statement actually adds the android plugin to the Gradle DSL.

dependencies

The dependencies block in the buildscript block specifies dependencies for the build script. The dependencies block 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.

dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
}

The compile fileTree(dir: 'libs', include: '*.jar') statement indicates that all JARs found in the project's libs/ directory should be included as part of the compile process. This is to support the classic way of building Android apps, with JARs in libs/.

android

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

While there are lots of possibilities for what can go in this block, the Eclipse-generated build.gradle has two statements (compileSdkVersion and buildToolsVersion) and a sourcesets block.

android {
    compileSdkVersion 18
    buildToolsVersion "19.0.1"

    sourceSets {
        main {
            ...
        }
    }
}
Gradle and Legacy Projects

```java
manifest.srcFile 'AndroidManifest.xml'
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')
}
```

compileSdkVersion replaces the target line in project.properties, or the build
target in Eclipse (Project > Properties > Android). Here, you specify the API level to
be compiled against, as a simple API level (e.g., 18). Note that if you are still using
Maps V1, or otherwise need to specify a build target that is not a pure API level, you
can pass the appropriate string to compileSdkVersion. For example, if your
project.properties file had target=Google Inc.:Google APIs:10 for Ant and
Eclipse, the equivalent in Gradle is compileSdkVersion "Google Inc.:Google
APIs:10".

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:

2137
Note that the SDK Manager will allow you to download the latest version of the Gradle build tools (appearing as 18.1 in the above screenshot) plus prior versions (18.0.1 and 17 in the above screenshot).

Beyond this, buildToolsVersion is presently undocumented. Therefore, it is unclear if this is a minimum value (and a newer set of tools will work) or if this is a specific value (and if you have a different set installed, your build will fail).

**sourcesets**

The bulk of the android-specific configuration in the exported build.gradle file comes in the sourceSets block inside the android block. Here, we specify what should be compiled.

Much of this is in the main block, defining a single set of sources:

```groovy
main {
    manifest.srcFile 'AndroidManifest.xml'
    java.srcDirs = ['src']
    resources.srcDirs = ['src']
    aidl.srcDirs = ['src']
    renderscript.srcDirs = ['src']
    res.srcDirs = ['res']
}
```
The new build system 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 rest of the sourceSets block consists of:

- Indicating where tests can be found, which is invalid in this case, as this project has no tests (this would appear to be a bug in the stock template).
- Indicating where details for specific build types could be found, for debug and release builds. This concept will be explored in greater detail in the next chapter.
Gradle and the New Project Structure

The preceding 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 the new build system'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.

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.

NOTE: This chapter discusses a pre-release technology. The technology is undergoing rapid revision and suffers from a dearth of documentation. More so than with other chapters in this book, you may encounter behavior that differs from what is documented. As the saying goes, “your mileage may vary”.

2141
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.
Source Sets

To quote the Gradle documentation: “A source set is simply a group of source files which are compiled and executed together.”

Source sets, on their own, have no particular semantic meaning. You can elect to have your project use a single source set, or several source sets, for organizing your code. You might have different source sets 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 source set, typically named `main`, but other features of the new build system will involve additional source sets.

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 Ant and 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 type, where you can indicate that the Linux `.so` files for a project should be compiled in debug mode.
GRADLE AND THE NEW PROJECT STRUCTURE

As we will see, creating a new build type involves modifications to the build.gradle file and adding a matching source set.

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.

As we will see, creating a new product flavor involves modifications to the build.gradle file and adding a matching source set.
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 Groups

Sometimes, even this is insufficient flexibility.

For example, you might make heavy use of the NDK, directly or indirectly (e.g., using SQLCipher for Android). Normally, you would build a single APK that contains compiled NDK binaries for all CPU architectures that you wish to support. However, that can substantially increase the size of your APK, and if you are bumping up against the file size limit of a distribution channel (e.g., the 50MB limit of the Play Store), you might want to consider separate APKs per architecture. Notably, the Play Store supports this, allowing you to upload one APK per architecture for the same product (with the same package name), and the Play Store infrastructure will deliver the right APK to the device based upon its CPU type.

You could use product flavors to arrange to have separate builds for the different CPU architectures, to support this process. However, you might also want to have different output for other reasons, 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 group”. However, you can organize your flavors into your own separate flavor groups (e.g., one for CPU architecture, one for distribution channel).

These then add another factor into the cross-product that determines your build variants. Suppose we have a cpu flavor group, consisting of arm and x86 product flavors, and we have a channel flavor group, 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 + arm
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.

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. Both of these limitations are likely to be lifted in the future.

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 is a clone of the Eclipse-generated “Hello, World” sample seen in the previous chapter, but reorganized to follow the new project structure.
The Directory Tree

The original sample directory tree is fairly conventional, just with some added Gradle-specific files:

Hello
|-- AndroidManifest.xml
|-- assets/
|-- build.gradle
|-- libs/
| |-- android-support-v4.jar
|-- local.properties
|-- 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)
GRADLE AND THE NEW PROJECT STRUCTURE

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
│   │   ├── 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
```

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 source set, 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

You might think that the build.gradle file would be more complicated. In reality, it is simpler, because a lot of the complexity of the original sample was because we were retrofitting the legacy project structure. So, we are left with:

```groovy
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.8.+'
    }
}
apply plugin: 'android'

dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
}

android {
    compileSdkVersion 18
    buildToolsVersion "19.0.1"
}
```

We still have the buildscript block to describe what we need for our build tools, the android plugin, a pointer to our libs/ directory, 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 same tasks as before, including installDebug but still not an installRelease task.

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.

**Source Set**

Each build type can have its own associated source set. If you skip the directory for it, that means that the build type is not contributing changes to the main source set.

So, in this sample project, we have a replacement version of the `strings.xml` resource, in a debug source set:

```plaintext
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/
```
GRADLE AND THE NEW PROJECT STRUCTURE

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
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">HelloGradle DEBUG</string>
</resources>
```

As we will see, resources in build types’ source sets 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 `sourcesets` block 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:

```gradle
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath 'com.android.tools.build:gradle:0.9.0'
  }
}
apply plugin: 'android'

dependencies {
  compile fileTree(dir: 'libs', include: '/*.jar')
```

---

Subscribe to updates at https://commonsware.com

Special Creative Commons BY-NC-SA 4.0 License Edition
The defaultConfig block allows us to change aspects of what is found in the AndroidManifest.xml file, replacing anything found in the actual file from the main source set.

Notable properties that we can specify include:

- `versionCode` (to override android:versionCode in the root `<manifest>` element)
- `versionName` (to override android:versionName in the root `<manifest>` element)
- `minSdkVersion` (to override android:minSdkVersion from the `<uses-sdk>` element)
- `targetSdkVersion` (to override android:targetSdkVersion from the `<uses-sdk>` element)
- `packageName` (to override package in the root `<manifest>` element)
The `buildTypes` block is where we configure the behavior of the build types. Each build type to be configured gets its own block 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)
- `packageNameSuffix` (to append to the package name specified by the manifest or the `defaultConfig` `packageName` property)
- `versionNameSuffix` (to append to the version name specified by the manifest or the `defaultConfig` `versionName` property)

Our debug build type adds suffixes to the version name and the package name. Note that altering the package name 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 `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.

We can also have a `signingConfig` property, referencing the name of a signing configuration specified in the `signingConfigs` block. This is used to provide rules for how to sign the APK that is assembled by Grade. In this project's `build.gradle` file, we have a `release` block 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` block in `buildTypes` references the signing configuration we want as `signingConfigs.release`. All of these Groovy blocks 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 block.

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 this StackOverflow question.

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.

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 source set 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

Resources from the build type's source set are merged into the resources from the main source set, 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 source set, but if there is a collision, the result is a build error. Either the build type or the main source set 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, most likely you will need to define that class in all build types, so that any references from main to that class are satisfied for all build types.

**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 source sets, 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` block in `build.gradle`:

```groovy
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.9.0'
    }
}
apply plugin: 'android'

dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
}

android {
    compileSdkVersion 18
    buildToolsVersion "19.0.1"

    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'
        }
    }
}
```
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` block 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.
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 source set. 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/
In the source sets, 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 source set, to delegate the handling of onOptionsItemSelected():

```java
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);
    }

```
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:

```java
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 block in the build.gradle file:

```gradle
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.9.0'
    }
}
apply plugin: 'android'

dependencies {
    compile fileTree(dir: 'libs', include: '*.*')
}
android {
```
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 packageName, as is done in this build.gradle file.
In terms of order of precedence:

- Product flavors override the main source set 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

Revisiting the Legacy Gradle File

Given all of this, let’s revisit the original build.gradle file, exported from Eclipse:

```gradle
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.9.0'
    }
}
apply plugin: 'android'

dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
}

android {
    compileSdkVersion 18
    buildToolsVersion "19.0.1"

    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
        }
    }
}
```
The main block in the sourceSets block overrides the default locations of files that comprise the source set. For example, in the new project structure, the Java source code goes in java/ within the source set, 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 source sets, if they exist, should be in a separate build-types/ directory, as the normal src/ location is being used for Java source code.

Discussion of the instrumentTest is covered in the chapter on Gradle and testing.
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.

NOTE: This chapter discusses a pre-release technology. The technology is undergoing rapid revision and suffers from a dearth of documentation. More so than with other chapters in this book, you may encounter behavior that differs from what is documented. As the saying goes, “your mileage may vary”.
“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.

The Dependencies Block… and the Other Dependencies Block

A build.gradle file is likely to have two dependencies blocks.

One will be inside the buildscript block, 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 block 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
GRADLE AND DEPENDENCIES

- 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` block that looks like this:

```groovy
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.

The 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.
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.

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 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.

### Creating a 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 android plugin, while Android library projects use the android-library plugin:
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.9.0'
    }
}

apply plugin: 'android-library'

android {
    compileSdkVersion 18
    buildToolsVersion "19.0.1"
}

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 will is covered in the chapter on Gradle and testing

Depending Upon the Library Project

Unfortunately, depending upon an Android library project is a bit more complex than before.

In most cases, you will need to 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.

You can skip that step if you can reorganize your code base into a project and set of sub-projects, which is covered in the next section.

Depending Upon Sub-Projects

In classic Ant/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:
**Gradle and Dependencies**

- 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):

```groovy
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.

To indicate that HelloLibraryConsumer wishes to consume code from the HelloLibrary library, another line is added to the dependencies block, referencing the sub-project:

```groovy
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.9.0'
    }
}
apply plugin: 'android'

dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
    compile project(':libraries:HelloLibrary')
}
android {
    compileSdkVersion 18
}
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.

**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.

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 block 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 block. 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 `com.commonsware.cwac:colormixer:0.5` 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

### Depending Upon Maven Central Artifacts

The single most common place to get artifacts is [Maven Central](https://mvnrepository.com). 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 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` block, at the top level (i.e., distinct from the one inside the `buildscript` block), that requests `mavenCentral()`:

```
repositories {
  mavenCentral()
}
```

Then, you can have `compile` statements in your `dependencies` block 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](https://mvnrepository.com) Web site”, where you can type in the name of a popular library (e.g., `otto`) and get the corresponding dependencies block:

```
dependencies {
  compile 'com.squareup:otto:1.3.4'
}
```

If you have several dependencies, list them in the one `dependencies` block, 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')
}
```
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.

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 block.

There are several artifacts of note in these repositories, with the versions current as of this writing shown:

- com.android.support:support-v4:19.0.0 and com.android.support:support-v13:19.0.0 for the -v4 and -v13 editions of the Android Support package
- com.android.support:appcompat-v7:19.0.0 for the AppCompat action bar backport
• com.android.support:gridlayout-v7:19.0.0 for the GridLayout backport
• com.google.android.gms:play-services:4.0.30 for the Play Services SDK

By referencing these artifacts, you no longer need to mess around with copying JARs or attaching Android library projects to your own projects.

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:

```groovy
repositories {
    maven {
        url "https://repo.commonsware.com"
    }
}
```

This repositories block adds a Maven-style repository, located at http://repo.commonsware.com.

```groovy
repositories {
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}
```

This is an equivalent repositories block, 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.

Consuming artifacts from your local repository is just a matter of having a mavenLocal() entry in your repositories block:

```groovy
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:

```groovy
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 android-maven-plugin, which, as the name suggests, is a plugin for Gradle that improves the publishing of Maven artifacts from Android builds.

To use this plugin, add classpath 'com.github.dcendents:android-maven-plugin:1.0' to your buildscript dependencies block in your Android library project’s build.gradle:

```groovy
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.github.dcendents:android-maven-plugin:1.0'
    }
}
```

Then, after the apply plugin: 'android-library' statement, add apply plugin: 'android-maven'. At this point, the gradle install command will build your AAR and deploy it to your local Maven repository, where it can be picked up by apps that have the mavenLocal() repository configured in their own build.gradle files.

However, for this to work, you will need to tailor the group, artifact ID, and version for your library. The version and group are configured via simple statements at the top level of your build.gradle file for the library project:

```groovy
version '0.5'
group 'com.commonsware.cwac'
```

The artifact ID is determined by default from the name of the directory that contains the library project. So a library project in a colormixer/ directory will be given the artifact ID of colormixer.

### 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.

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 StackOverflow question providing a quick-and-dirty bit of Gradle code to add JAR-creation tasks to an android-library project:

```groovy
android.libraryVariants.all { variant ->
    def name = variant.buildType.name
    if (name.equals(com.android.builder.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.

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 block, 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 block 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 block after you define the build type in the build.gradle file.
after Gradle has defined your build type will your custom compile statement be available.

There is also instrumentTestCompile, which defines dependencies solely for use with testing. This is covered in greater detail in the next chapter.

**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:

```groovy
productFlavors {
  standard {
    packageName "com.commonsware.android.wearable.qr"
  }
  imwatch {
    packageName "com.commonsware.android.wearable.qr.imwatch"
  }
  sony {
    packageName "com.commonsware.android.wearable.qr.sony"
  }
}
```

A dependencies block *after* the android block containing the above productFlavors configuration could have a mix of per-flavor dependencies and flavor-specific dependencies, such as:

```groovy
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.

## 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:

```ini
# 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):
```
GRADLE AND DEPENDENCIES

#proguard.config=${sdk.dir}/tools/proguard/proguard-android.txt:proguard-project.txt

# Project target.
target=android-17
android.library.reference.1=../layouts

For the Gradle build, the demo version of build.gradle depends upon the layouts AAR file, published as a Maven artifact to the CommonsWare artifact repository:

buildscript {  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        classpath 'com.android.tools.build:gradle:0.8.+'  
    }  
}  
apply plugin: 'android'

repositories {  
    maven {  
        url "https://repo.commonsware.com.s3.amazonaws.com"  
    }  
}  

dependencies {  
    compile 'com.commonsware.cwac:layouts:0.3.0'  
}  

android {  
    compileSdkVersion 17
    buildToolsVersion "19.0.1"
    
    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')
    }
}
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:

```groovy
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.8.+'
        classpath 'com.github.dcendents:android-maven-plugin:1.0'
    }
}
apply plugin: 'android-library'
apply plugin: 'android-maven'
version '0.3.0'
group 'com.commonsware.cwac'
dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
}
android {
    compileSdkVersion 17
    buildToolsVersion "19.0.1"
    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.BuilderConstants.DEBUG)) {
        return; // Skip debug builds.
    }
    def task = project.tasks.create "jar${name.capitalize()}", Jar
```

---

2181
GRADLE AND DEPENDENCIES

```
task.dependsOn variant.javaCompile

task.from variant.javaCompile.destinationDir

// artifacts.add('archives', task);
```

It uses the android-maven plugin to enable the install 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 Maven repository. Separately, the author has a script that will push the necessary files to the Amazon S3-hosted CommonsWare Maven repository.

The library's build.gradle file also contains the custom Gradle code that adds a jarRelease task, as described earlier in this chapter. Hence, when the author wishes to push a new version of the library, the steps are:

1. Modify the version property to the proper value (shown here as 0.3.0)
2. Run gradle install 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). And, as before, for the purposes of an Ant/Eclipse build, the demo project has a project.properties file that adds a dependency upon the presentation library project:

```
# This file is automatically generated by Android Tools.
# Do not modify this file -- YOUR CHANGES WILL BE ERASED!
#```

2182
GRADLE AND DEPENDENCIES

# 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 project is the same as the one for the
CWAC-Layers:

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.9.+'
    }
}

apply plugin: 'android'

repositories {
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}

dependencies {
    compile 'com.commonsware.cwac:presentation:0.4.+'
}

android {
    compileSdkVersion 17
    buildToolsVersion "19.0.1"

    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            aidl.srcDirs = ['src']
            renderscript.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }
    }
}
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 block, supplying the URL to the CommonsWare Maven repository
2. It adds a dependency on the CWAC-Layouts library in its dependencies block

```
repositories {
    maven {
        url "https://repo.commonsware.com.s3.amazonaws.com"
    }
}

dependencies {
    compile 'com.commonsware.cwac:layouts:0.3.+'
    compile fileTree(dir: 'libs', include: '*.jar')
}
```

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.
Of course, it is always nice when these apps that we write actually work as expected.

Android has a variety of tools available for testing, and these are incrementally moving into the Gradle for Android arena. In this chapter, we will examine how to set up JUnit-style instrumentation tests in Gradle for Android and how Gradle for Android improves upon what we have had before. Also, we will examine how other testing frameworks can be employed from Gradle for Android.

**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. Having read the chapter on Gradle dependencies would also be a pretty good idea.

Since this chapter also covers testing, the chapters on testing are also prerequisites:

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

**NOTE:** This chapter discusses a pre-release technology. The technology is undergoing rapid revision and suffers from a dearth of documentation. More so than with other chapters in this book, you may encounter behavior that differs from what is documented. As the saying goes, “your mileage may vary.”
JUnit/Instrumentation Testing

Android has supported “instrumentation testing” from the outset. The primary way we employ instrumentation testing is via JUnit-based unit tests. We would write JUnit test suites and run them with a JUnit test runner, either from an IDE or from the command line.

In many respects, that has not changed.

However, Gradle for Android not only provides a simplified way to execute a test suite from the command line, it also:

- Makes it easier for us to write these tests, by dumping the dedicated test project in favor of a special source set
- Generates more readable output than running instrumentation tests through the stock Ant task
- Adds other benefits, such as “going wide” and testing on multiple devices or emulators simultaneously

Testing Applications

Most of the time, we are testing Android application projects. And, much of the time, those projects stand alone or are tied to dependencies that we did not write. In those cases, instrumentation testing follows a fairly standard recipe, covered in this section. Later sections will cover testing Android library projects.

The androidTest Source Set

We used to have a dedicated test project for our instrumentation tests, where the code for those tests would reside.

Now, we have a dedicated source set 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 source set, androidTest can have Java code, resources, etc. It does not need an AndroidManifest.xml file, though, as that will be auto-generated.
The Gradle/HelloInstrumentTest sample project contains an androidTest source set with a single file, java/com/commonsware/android/gradle/hello/SillyTest.java, containing the same SillyTest as was seen in the chapter on JUnit and Android:

```java
package com.commonsware.android.gradle.hello;

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 {
        super.tearDown();
        // do termination here, run on every test method
    }

    public void testNonsense() {
        assertTrue(1==1);
    }
}
```

In your own androidTest, you can have whatever JUnit TestCase classes you like, using all the same capabilities that you used in crafting JUnit test cases in a separate project. If, however, your JUnit test code relies on third-party libraries, you will need to configure those dependencies; this is covered later in this chapter.

The Gradle Configuration

Your defaultConfig block will need to define a couple of properties for the testing to work:

- `testPackageName` is the name of the package to be used for the instrument test APK. For testing apps, this needs to be a separate package name from the package name of the app.
- `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.
For example, the build.gradle file for the HelloInstrumentTest project contains both of these properties:

```gradle
buildscript {
    repositories {
        mavenCentral() 
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.9.0'
    }
}
apply plugin: 'android'

dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
}

android {
    compileSdkVersion 18
    buildToolsVersion "19.0.1"

    defaultConfig {
        testPackageName "com.commonsware.android.gradle.hello.test"
        testInstrumentationRunner "android.test.InstrumentationTestRunner"
    }
}
```

## The Gradle Tasks

The primary 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 source set).

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.

Hence, running `gradle connectedCheck` offers a few conveniences over the equivalent `ant test` command:

- It runs tests in parallel, across multiple devices/emulators, where available
- It cleans up after itself, by uninstalling the apps
The Test Results

Raw test results, in XML format, will be written to build/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/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.

Testing Library Projects

Before Gradle, to test an Android library project, you would create a separate test project and set it up to “test itself” (i.e., the package being tested was the test project’s own package). You could then add the Android library project to the test project and write test cases to exercise the library.

What Gradle for Android gives us is a bit of a “mashup” between the original approach and what we have with testing regular apps using Gradle.

A Gradle-built Android library project can have an androidTest source set, 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 source set 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.

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 supports an androidTestCompile statement in the dependencies block. Where compile states dependencies for the entire project; androidTestCompile states dependencies only for the instrumentation testing:

```
dependencies {
    // regular dependencies
    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.

### Testing Legacy Project Structures

Suppose that you have a project that you still need to be buildable using Eclipse or Ant, 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 source set, as Ant and Eclipse do not know what a “source set” even is.

However, just as you can teach the main source set where it can find its files, you can do the same thing with the androidTest source set.

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 block 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']
        }
    }
}
```
GRADLE AND TESTING

```java
renderscript.srcDirs = ['tests/src']
res.srcDirs = ['tests/res']
assets.srcDirs = ['tests/assets']
```}

(from Xavier Ducrohet's answer on StackOverflow)

Here, we teach both the main and the androidTest source sets where their respective files are. In the case of the androidTest source set, we point it to files in the tests/subdirectory.

Given this, Gradle can run the tests, just as Ant and Eclipse can.

Other Types of Testing

While JUnit-style instrumentation testing is the bedrock of many test plans, it is far from the only way of testing Android applications. Some of the alternatives are also available for use from Gradle for Android, though not necessarily by officially-supported means.

Notably:

- Novoda offers a [Gradle plugin](https://commonsware.com) for exercising an application using monkeyrunner
- Aaron Newell offers a [Gradle recipe](https://commonsware.com) for launching uiautomator tests from Gradle
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.

**NOTE**: This chapter discusses a pre-release technology. The technology is undergoing rapid revision and suffers from a dearth of documentation. More so than with other chapters in this book, you may encounter behavior that differs from what is documented. As the saying goes, “your mileage may vary”.

## Supporting AIDL

If your application is using AIDL, to export a service to third parties or to consume a service from third parties, the default location for your AIDL files is a dedicated `aidl/` directory, as a peer of your `java/` and `res/` directories in your source set.
However, you can configure Gradle for Android to pull your AIDL files from a different location by configuring the aidl.srcDirs of a source set, such as the main source set.

You can see this in the Gradle/HelloAIDL sample project. This is a clone of one of the AIDL sample apps from the chapter on binding and remote services.

This project consists of two sub-projects: one for the remote service and one for its client. However, they both share a common AIDL definition. Rather than having two copies of the AIDL or relying on OS-level tricks like symlinks, we have an aidl/ directory in the project’s root, outside either sub-project. That aidl/ directory then has the classic set of sub-directories based upon the Java package name used in the AIDL, leading eventually to the .aidl file.

The build.gradle file for each of the sub-projects then configures the main sourceset to point to this common AIDL directory, as can be seen in the Client/sub-project’s build.gradle file:

```groovy
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.9.0'
    }
}
apply plugin: 'android'

dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
}

android {
    compileSdkVersion 18
    buildToolsVersion "19.0.1"

    sourceSets {
        main {
            aidl.srcDirs=[rootProject.file('aidl')]
        }
    }
}
```

Here, rootProject.file('aidl') means “use the root project’s location to derive the location of a file named aidl”. Otherwise, since this code resides in a sub-
Advanced Gradle for Android Tips

project, we would need to use `file('../aidl')`, which works, but is less flexible in the face of possible project reorganization.

The fact that we have AIDL files in an AIDL source directory provides enough information to the Android build tools to run those files through the AIDL processor, to generate the Java code for our client proxy and interface stub, and to make those files available to the rest of the compile process.

Supporting 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 source set. 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 source set:

```
ConstantsSecure
  |—— build.gradle
  |—— libs
  |   |—— commons-codec.jar
  |   |—— guava-r09.jar
  |   |—— sqlcipher.jar
```
(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 source set.

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 source set 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 block 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 blocks 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:0.6.2-SNAPSHOT'
  }
}
apply plugin: 'android'
android {
  compileSdkVersion 15
```
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 source sets, such as the main/ source set. However, you can override that using jni.srcDirs in your configuration of a source set.

For example, let us take a peek at the CWAC-AndDown project.

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 that take in Markdown and want to render the results in a WebView (though you can use Html.fromString() to put the results in a TextView if you want).
Since CWAC-AndDown uses hoedown's C code, it has a \texttt{jni/} directory containing the JNI wrapper C code, the appropriate makefiles, plus a \texttt{src/} subdirectory containing the hoedown source code.

The classic way to build the project was to manually run the \texttt{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 \texttt{jni/} directory in its original spot, alongside the \texttt{res/}, \texttt{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 \texttt{jni.srcDirs} for our main sourceset in \texttt{build.gradle}:

```groovy
sourceSets {
  main {
    manifest.srcFile 'AndroidManifest.xml'
    java.srcDirs = ['src']
    resources.srcDirs = ['src']
    aidl.srcDirs = ['src']
    renderscript.srcDirs = ['src']
    res.srcDirs = ['res']
    assets.srcDirs = ['assets']
    jni.srcDirs = ['jni']
  }

  debug.setRoot('build-types/debug')
  release.setRoot('build-types/release')
}
```

The CWAC-AndDown \texttt{build.gradle} file also contains the \texttt{ndk} block 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.
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. Mostly, it involves adding the following to the bottom of your `build.gradle` file:

```groovy
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:

```
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.

**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

**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 project.ext object via a block. Xavier Ducrohet of Google demonstrated this with an example of consolidating your compileSdkVersion across sub-projects. In the top-level build.gradle file, you would put common values to be used:
Sub-projects can then reference `rootProject.ext` to retrieve those values:

```groovy
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` block. This contains bits of configuration that will be applied to each of the subprojects as a part of their builds.

The `HelloAIDL` sample project mentioned earlier in this chapter demonstrates this. The `build.gradle` in the overall project root (outside the `Client/` and `Service/` sub-projects) has a `subprojects` block to define the code-signing rules for these two applications:

```groovy
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.9.0'
    }
}

subprojects {
    apply plugin: 'android'

    android {
        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 block contains its own reference to the android plugin for Gradle, in addition to android block 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 android-library plugin, while the application sub-project needs to configure the android plugin. The subprojects block is only good for common configuration to apply to all sub-projects regardless of project type.

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.

So, to achieve the synchronized compileSdkVersion value, you could have a gradle.properties file with:

```
COMPILE_SDK_VERSION=19
```

Then, your sub-projects’ build.gradle files could use:

```
android {
    compileSdkVersion COMPILE_SDK_VERSION
}
```

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.

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:
Here, we are defining a build tools version for use with the buildToolsVersion property in the android block. The sub-projects use the BUILD_TOOLS_VERSION property that we defined in gradle.properties in their own build.gradle files, such as the one from the Client/ sub-project:

```gradle
buildscript {
    repositories { 
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.9.0'
    }
}
apply plugin: 'android'
dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
}
android {
    compileSdkVersion 18
    buildToolsVersion BUILD_TOOLS_VERSION
    sourceSets {
        main {
            aidl.srcDirs=[rootProject.file('aidl')]
        }
    }
}
```

With both sub-projects referencing their build tools version from the common gradle.properties file, we can change the version once and have it reflect automatically in both sub-projects.

**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.
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:

```groovy
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.9.0'
    }
}

subprojects {
    apply plugin: 'android'

    android {
        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']
                }
            }
        }
    }
}
```

Let’s look at the key lines, one at a time:

```groovy
def signingPropFile = rootProject.file('signing.properties')
```
This statement grabs the signing.properties file from the root project and assigns it to the signingPropsFile variable. Groovy, by default, is a dynamic language and does not use data types for its variables. Under the covers, signingPropsFile is a java.io.File object, just like you are used to in ordinary Java/Android development.

```groovy
if (signingPropsFile.canRead()) {
}
```

Since signingPropsFile is a File, we can call a canRead() method to confirm that the file exists and is readable.

```groovy
def Properties signingProps = new Properties()
```

This creates an empty instance of a java.util.Properties object and assigns it to the signingProps variable.

```groovy
signingProps.load(new FileInputStream(signingPropsFile))
```

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.

```groovy
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.

### 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:

```groovy
def versionPropsFile = file('version.properties')

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 block.

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.

**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, a StackOverflow user named codaRoy 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 block:

```groovy
buildTypes {
    debug {
        packageNameSuffix ".d"
        versionNameSuffix "-debug"
    }

    release {
        signingConfig signingConfigs.release
    }

    mezzanine.initWith(buildTypes.release)

    mezzanine {
        packageNameSuffix ".mezz"
        debuggable true
        signingConfig signingConfigs.release
    }

    applicationVariants.all { v ->
        def f = v.outputFile
        def fname = f.name.replace(".apk",
            "-${defaultConfig.versionName}.apk")
    }
}
When defining the build types, we iterate over all application variants (build type/product flavor combinations) and modify the `outputFile` property of the variant to embed the `defaultConfig.versionName` value.

### Adding to BuildConfig

The Android development tools have been code-generating the `BuildConfig` class for some time now. The sole element of that class is the `DEBUG` flag, 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.

However, you can add your own data members to `BuildConfig`, by including a `buildConfigField` statement in the `defaultConfig` block of your `android` block:

```java
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` block defines three different variations of a `SERVER_URL` field on the `BuildConfig` object:

```java
buildTypes {
  debug {
    packageNameSuffix ".d"
    versionNameSuffix "-debug"
    buildConfigField "String", "SERVER_URL", "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 {
    packageNameSuffix ".mezz"
    debuggable true
    signingConfig signingConfigs.release
    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 block, to have a different server URL for mezzanine builds than we use for release or debug builds.
Trail: Other Tools
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 Eclipse, those two steps can even be combined — Eclipse 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.

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
Advanced Emulator Capabilities

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 551: SDK Manager, Showing “Intel x86 Atom System Image”](image)

When you download that, the next time you choose API Level 15 for an AVD, you will have an option of CPU architecture:
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:
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:
Advanced Emulator Capabilities

In the dialog that appears, choose “GPU Emulation” in the drop-down:

![AVD Configuration Editor, With “New…” Button Focused](image)

**Figure 554: AVD Configuration Editor, With “New…” Button Focused**

Then click OK, which will add “GPU Emulation” to the table:

![AVD Hardware Options Dialog](image)

**Figure 555: AVD Hardware Options Dialog**
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:
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:
Figure 558: AVD Manager, Device Definitions Tab

That will bring up an empty device definition 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
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.
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 Eclipse and available from the command line, for reporting similar sorts of issues with an Android project’s Java code, resources, and manifest.

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 Eclipse, these result in red “X” notations on the files in the Package Explorer, and frequently result in red squiggle 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**

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 560: Eclipse Lint Warnings View](image)

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 .:
```
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:

```bash
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.

What to Fix

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.

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:
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:
$ 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.

2228
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
<?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.
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. From the command line, run the monitor program to bring up the Android Device Monitor, choose Window > Open Perspective from the main menu, and open Hierarchy View
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:
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:
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
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:

```java
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.
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
**Using DDMS**

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.

![DDMS, with Emulator Selected](image)

*Figure 565: 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.
**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:

![DDMS Screen Capture Dialog](image)

*Figure 566: 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.
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:

![DDMS Emulator Control View](image)

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.
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 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.
JetBrains’ IntelliJ IDEA has long been an important counterweight to Eclipse in Java development, being popular enough to help prevent a slide into an IDE monoculture.

IDEA has supported Android application development for a while, and it is the IDE of choice among many Android “power developers”, both inside and outside of Google.

IDEA’s Android support comes in the form of a plugin, albeit one that ships standard with IDEA, including with the open source Community Edition. Google has taken over maintenance of that plugin, as part of its effort to create Android Studio, an IDEA-based Android-focused IDE.

This chapter will cover some of the more important operations in Android application development (outside of standard Java programming) and how you can accomplish those using IDEA, specifically IDEA Community Edition 12.1.

Prerequisites

Understanding this chapter requires that you have read the core chapters of the book.

Creating a New Project

When you first start up IDEA, by default you will be greeted with a dialog allowing you to open recent projects or perform some “quick actions”:
From here, you can choose “Create New Project”. Or, given an open IDE, you can use File > New Project... from the main menu to get to the same spot:
The list on the left indicates the type of “module” that you wish to create. To create a new Android application, choose “Application Module”.

You can then give the project:

- A name
- A location on the filesystem where the project files will be stored
- The Android build target
- Various IDEA-specific settings, such as the project file format

Clicking the “Next” button will allow you to configure additional Android-specific properties:
Specifically, you can choose:

- The application name (used for the `app_name` string resource)
- The Java package name (used for the `package` attribute in the root `<manifest>` element of the manifest)
- Whether or not you want a “Hello, world!” activity to be created (and, if you do, what the Java class name should be)
- How you want to run the project by default (e.g., show a device chooser dialog)

Clicking “Finish” will then create the empty project for you.

**Importing an Existing Project**

Of course, you may have existing projects, particularly if you have been doing development using Eclipse or something else previously and now wish to move to IDEA. You can import those projects into IDEA, much like how you import projects into Eclipse.
From the main menu, you can choose File> Import Project to begin importing a project. Or, you can choose “Import Project” from the startup dialog.

Either way, you will be presented with a typical browse-the-filesystem dialog, where you can pick the root directory of your project. Clicking OK will bring up an “Import Project” dialog:

![IDEA Import Wizard, Initial Page](image)

What happens now depends upon the build process for the project that you are importing. Here, we will examine the two most common options: Eclipse or a command-line build:

**Eclipse**

If the project that you are trying to import is an Eclipse project, choose Eclipse in the “Import Project” dialog and click “Next”. This will bring up the second page of the wizard:
The defaults here should be fine in many cases. Clicking “Next” brings up the third page of the wizard:
Here, you can choose the project(s) to import. Typically, there will be just one. Clicking “Next” will bring you to the fourth and final page of the wizard:
Here, you specify what the “project SDK”. This is not the build target, nor anything else seen in Eclipse. Instead, this appears to determine what set of tools are used for the build process, and seems to be independent of the build target in IDEA. It is not obvious what impact this has; in particular, choosing a plain Java (e.g., “Java 1.6”) or an Android SDK seems to have no apparent effect.

**Command Line**

The process for a command-line build is similar. However, instead of choosing Eclipse, you will need to toggle the radio button from “Import project from external model” to “Create project from existing sources”. Clicking “Next” will bring up a slightly different version of the second page, where you can only specify the name and location of the project:
Clicking “Next” here will bring up another version of the third wizard page, where you indicate which directories contain files of relevance to your project — most likely, you will accept the defaults:

*Figure 575: IDEA Import Wizard, Second Page, Command-Line Build*
Clicking “Next” here will bring up the contents of your `libs/` directory, if any, that you can choose to keep:
Clicking “Next” here will bring up a wizard page for you to “review [the] suggested module structure for the project”:
Clicking “Next” here will bring up the wizard page that matches the fourth wizard page of the Eclipse import, where you can specify the “project SDK”:
One more click of “Next” brings up a wizard page showing “frameworks” that were “detected” by the import process. An Android project should show up, along with a reference to the manifest:
Finally, clicking “Finish” will import the project.

**Attaching a JAR**

In Eclipse, adding a JAR is a matter of copying the JAR to the project’s `libs/` directory (and, on the R22 version of the Android Developer Tools or higher, confirming the “Order & Export” portion of your project build path).

In IDEA, the equivalent is to copy the JAR into `libs/`, then ensure that `libs/` is listed as a dependency. To do this:

- Choose File > Project Structure from the main menu, to bring up the “Project Structure” dialog:
Choose Modules in the “Project Settings” list on the left:
**Figure 58: IDEA Project Structure Dialog, Showing Modules**

- Click on the Dependencies tab (shown selected above)
- Click on the + icon to the left of the list of dependencies, to show a small fly-out menu, and choose “Jars or directories” from there to bring up a directory-chooser dialog
- In the chooser, click on your project’s `libs/` directory, then click OK to close that dialog
- Click OK on the “Project Structure” dialog to close it

At that point, your JAR is available for use.

**Accessing Android Tools**

The Tools > Android option in the main menu will give you access to many common Android tools, including:

- Monitor (for access to DDMS and Hierarchy View)
- Draw 9 Patch
- AVD Manager
- SDK Manager
Note, though, that once you open the AVD Manager this way, you cannot close it without closing the IDE itself. You may wish to open the AVD Manager from the command line instead (`android avd`).

![IDEA Android Tools Menu](image)

*Figure 583: IDEA Android Tools Menu*

To access LogCat, look for a minimized docked view labeled Android:

![IDEA Android View, Minimized](image)

*Figure 584: IDEA Android View, Minimized*

Tapping on it will open it up to show you LogCat messages:

![LogCat in IDEA](image)

*Figure 585: LogCat in IDEA*
Run and Debug a Project

Much as with Eclipse, there are “run” and “debug” toolbar buttons in IDEA to run or debug your project.

Also, much as with Eclipse, there are “run configurations” that indicate what should happen when you run or debug a project. If you created a new project from scratch, the new-project wizard asked what behavior you would like (e.g., bring up a device chooser). If you imported a project, though, IDEA made a default selection, which may require adjustment.

To change run configurations, click the drop-arrow next to your project name in the toolbar:

![IDEA Project Drop-Down](image)

*Figure 586: IDEA Project Drop-Down*

From there, choose “Edit Configurations” to bring up the “Run/Debug Configurations” dialog:
Of note, the “Target Device” group can be set to “Show chooser dialog”, “USB device” (to always prefer hardware), or “Emulator” (to always prefer an emulator). In the latter case, you can specifically state an emulator AVD to launch, if it is not already running. Beyond that, the behavior of this dialog is analogous to its Eclipse equivalent.

### Editing Android-Specific Files

In Eclipse, editing Android-specific XML files brings up dedicated editors, with sub-tabs to allow you to edit the raw XML. In IDEA, for most files, including the manifest and most resources, opening the file only gives you an XML editor.

The exception is opening a layout resource, which brings up the IDEA graphical layout editor:
The contents of IDEA’s graphical layout editor is equivalent to Eclipse’s, albeit organized slightly differently:

- The widget-and-container palette is docked on the right by default, and the categorization is via checkboxes rather than accordion-style headers
- The tree of widgets and containers is docked in the upper-left corner by default
- The properties of a selected widget or container is docked in the lower-left corner by default
- The raw XML is available from a “Text” sub-tab, rather than a sub-tab named the same as the layout file itself

Otherwise, you will find the behavior of the graphical layout editor to be reasonably similar, with a few platform-specific differences to reflect the preferred styling of IDEA versus Eclipse.
IDEA-Specific Files

IDEA uses an `out/` directory where Eclipse and Ant use a `bin/` directory for the compiled output. Hence, you will find your APK files in there.

IDEA also creates:

- An `.iml` file in the root of your project, representing this IDEA module
- An `.idea` hidden directory in your project, containing a substantial number of configuration files

Using MAT

In Eclipse, you have the option of installing MAT as a plugin. With that installed, capturing and examining a heap dump can be accomplished completely within the IDE:

- Use DDMS to capture the heap dump
- Use MAT to examine the heap dump (opened automatically once the dump is captured in many circumstances)

In IDEA, you need to use the external Android Debug Monitor tool to capture the heap dump. The Monitor is available from the Tools > Android menu. There, you will see views matching DDMS from inside of Eclipse, so you can choose your process and dump its heap as before, albeit to a file on your filesystem.

To view the heap dump, though, you will need to install the standalone “RCP” version of MAT, that runs outside of Eclipse.
Trail: Production
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.
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, whether through Ant or Eclipse, creates a debug key for you automatically. That key is automatically applied when you create a debug version of your application (e.g., `ant debug` or `ant install`). 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 problem #3 from the previous section, 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).

So, developing in debug mode is easy. It is mostly when you move to production that things get a bit more interesting.

Creating a Production Signing Key

To 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`:

```
mmurphy@opti755:~$ keytool -genkey -v -keystore cw-release.keystore -alias cw-release -keyalg RSA -validity 10000
```

Figure 589: Running keytool
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)

The length of the validity 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. 10,000 days, presumably, is beyond the
expected lifespan of this signing mechanism. Also, the Play Store requires your key
to be valid beyond October 22, 2033.

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
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?
[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 1,024 bit RSA key pair and self-signed certificate (SHA1withRSA) 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):
Re-enter new password:
[Storing cw-release.keystore]
```

Figure 590: Results of running keytool
You will note that this is a self-signed certificate — 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.

In theory, you only need to do the above steps once per business.

**Signing with the Production Key**

To sign an application with a production key, you must first create an unsigned version of the APK. By default (e.g., `ant debug`), you get an APK signed with the debug key. Instead, specifically build a release version (e.g., `ant release`), which should give you an `-unsigned.apk` file in your project's `bin/` directory.

Next, to apply the key, you will use the `jarsigner` tool. Like `keytool`, `jarsigner` comes with the Java SDK, and so you should already have it on your development machine.

Here is an example of running `jarsigner`:

```
mmurphy@opti255:~/stuff/CommonsWare/projects/vidtry$ jarsigner -verbose -keystore -/cw-release.keystore -e ~/cw-release.keystore/bin/vidtry-unsigned.apk cw-release
```

**Figure 591: Running jarsigner**

In this case, the parameters supplied are:

1. `-verbose`, to explain what is going on as the program runs
2. `-keystore`, to indicate where the keystore that contains the production key resides (`~/cw-release.keystore`)
3. the path to the APK to sign (`bin/vidtry-unsigned.apk`)
4. the alias of the key in the keystore to apply (`cw-release`)

At this point, `jarsigner` will prompt you for the keystore's password (and the key's password if you supplied a distinct password for it to `keytool`), then it will apply the signature:
Next, you should test the signature by jarsigner -verify -verbose -certs on the same APK file, which now has a signature. You will get output akin to:

```
1090 Sat Aug 08 13:56:38 EDT 2009 META-INF/MANIFEST.MF
1211 Sat Aug 08 13:56:38 EDT 2009 META-INF/CW-RELEA.SF
946 Sat Aug 08 13:56:38 EDT 2009 META-INF/CW-RELEA.RSA
sm 1683 Sat Aug 08 13:54:46 EDT 2009
res/drawable/btn_media_player.9.png
   X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC",
   L=Unknown, ST=PA, C=US
   [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]
sm 743 Sat Aug 08 13:54:46 EDT 2009
res/drawable/btn_media_player_disabled.9.png
   X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC",
   L=Unknown, ST=PA, C=US
   [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]
sm 1030 Sat Aug 08 13:54:46 EDT 2009
res/drawable/btn_media_player_disabled_selected.9.png
   X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC",
   L=Unknown, ST=PA, C=US
   [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]
sm 1220 Sat Aug 08 13:54:46 EDT 2009
res/drawable/btn_media_player_pressed.9.png
   X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC",
   L=Unknown, ST=PA, C=US
   [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]
sm 1471 Sat Aug 08 13:54:46 EDT 2009
res/drawable/btn_media_player_selected.9.png
```

Figure 592: Results of running jarsigner
SIGNING YOUR APP

X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC",
L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]
sm 576 Sat Aug 08 13:54:46 EDT 2009
res/drawable/ic_media_pause.png

X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC",
L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]
sm 938 Sat Aug 08 13:54:46 EDT 2009
res/drawable/ic_media_play.png

X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC",
L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]
sm 1176 Sat Aug 08 13:54:46 EDT 2009
res/drawable/media_button_background.xml

X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC",
L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]

X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC",
L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]
sm 1368 Sat Aug 08 13:54:46 EDT 2009 AndroidManifest.xml

X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC",
L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]
sm 2888 Sat Aug 08 13:54:46 EDT 2009 resources.arsc

X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC",
L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]
sm 16860 Sat Aug 08 13:54:46 EDT 2009 classes.dex

s = signature was verified
m = entry is listed in manifest
k = at least one certificate was found in keystore

2270
In particular, you want to make sure that the name of the key is what you expect and is not “Android Debug”, which would indicate the APK was signed with the debug key instead of the production key.

At this point, you should also rename the APK, at least to remove the now-erroneous -unsigned portion of the filename.

Now, you have a production-signed APK, ready for distribution... or, hopefully, ready for more testing, then distribution.

**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 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.

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.
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 `android:versionCode` and `android:versionName` attributes in your `<manifest>` element in your `AndroidManifest.xml` file. The value of `android: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 `android:versionCode`, which 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.

**Package Name**

You also need to make sure that your package name — as denoted by the package attribute of the root `<manifest>` element — 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.

Since the manifest’s package name 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 package name 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](https://commonsware.com) 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:

```java
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 difficult 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(onBlooe);  

This will route all uncaught exceptions to an onBlooe handler:

```java
private Thread.UncaughtExceptionHandler onBlooe =
    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 API 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 Market 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... 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 other products or applications) then the terms of that separate agreement will govern your use of such information.

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.
Trail: Tuning Android Applications
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.
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
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.
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.
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,
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:

![DDMS Toolbar Button](image)

*Figure 593: 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).
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:
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:

```java
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);
    }
  }

  public static void main(String[] args) {
    StringConcatActivity activity = new StringConcatActivity();
    activity.onCreate();
    activity.onCreateSaveInstanceState();
    activity.performResume();
    activity.onConfigurationChanged(activity.getResources().getDisplayMetrics());
    activity.onLowMemory();
    activity.onSaveInstanceState(activity.getSavedInstanceState());
    activity.onResume();
    activity.onPause();
    activity.onPostCreate(activity);
    activity.onDestroy();
  }
}
```

2287
Here is a Traceview screen showing that code executed 100,000 times, as packaged in a StringPerfConcat activity in the Tuning/Traceview sample project:

![Traceview screen](image)

**Figure 595: 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.
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:

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:
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:

```java
package com.commonsware.android.traceview;

import android.view.View;
import android.widget.TextView;

public class StringBuilderActivity extends BaseActivity {
    public class StringBuilderTask extends BaseTask {
        protected String doTest() {
            StringBuilder result = new StringBuilder("This is a string");
            return result.toString();
        }
    }

    public class StringBuilderTask {
        public class StringBuilderTask {
            protected String doTest() {
                StringBuilder result = new StringBuilder("This is a string");
                return result.toString();
            }
        }
    }

    protected String doTest() {
        StringBuilder result = new StringBuilder("This is a string");
        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):

```
Figure 599: 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:
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:

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:
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.
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 “hiccup”s 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).
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.
**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.
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](https://commonsware.com)).

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:
```
** Finding CPU Bottlenecks **

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
- DrawDisplayList
  - Save
  - ClipRect
  - Translate
  - DrawText
  - RestoreToCount
- DrawPatch

2298
### Finding CPU Bottlenecks

<table>
<thead>
<tr>
<th>Caches:</th>
<th>Current memory usage / total memory usage (bytes):</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextureCache</td>
<td>1078032 / 25165824</td>
</tr>
<tr>
<td>LayerCache</td>
<td>7864320 / 16777216</td>
</tr>
<tr>
<td>GradientCache</td>
<td>0 / 524288</td>
</tr>
<tr>
<td>PathCache</td>
<td>0 / 4194304</td>
</tr>
<tr>
<td>CircleShapeCache</td>
<td>0 / 1048576</td>
</tr>
<tr>
<td>OvalShapeCache</td>
<td>0 / 1048576</td>
</tr>
<tr>
<td>RoundRectShapeCache</td>
<td>0 / 1048576</td>
</tr>
<tr>
<td>RectShapeCache</td>
<td>0 / 1048576</td>
</tr>
<tr>
<td>ArcShapeCache</td>
<td>0 / 1048576</td>
</tr>
<tr>
<td>TextDropShadowCache</td>
<td>0 / 2097152</td>
</tr>
<tr>
<td>FontRenderer 0</td>
<td>262144 / 262144</td>
</tr>
<tr>
<td>Other:</td>
<td></td>
</tr>
<tr>
<td>FboCache</td>
<td>3 / 16</td>
</tr>
<tr>
<td>PatchCache</td>
<td>89 / 512</td>
</tr>
<tr>
<td>Total memory usage:</td>
<td>9204496 bytes, 8.78 MB</td>
</tr>
</tbody>
</table>

Profile data in ms:

```plaintext
com.commonsware.android.anim.threepane/
com.commonsware.android.anim.threepane.MainActivity/
android.view.ViewRootImpl@4131e788
```

<table>
<thead>
<tr>
<th>Draw Process Execute</th>
<th>Draw</th>
<th>Process</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14.45</td>
<td>59.67</td>
<td>10.44</td>
</tr>
<tr>
<td></td>
<td>10.91</td>
<td>1.06</td>
<td>1.20</td>
</tr>
<tr>
<td></td>
<td>1.73</td>
<td>12.80</td>
<td>1.19</td>
</tr>
<tr>
<td></td>
<td>1.45</td>
<td>0.64</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td>2.15</td>
<td>0.47</td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td>0.79</td>
<td>0.50</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td>2.23</td>
<td>0.49</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>1.56</td>
<td>0.57</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>6.14</td>
<td>0.47</td>
<td>1.92</td>
</tr>
<tr>
<td></td>
<td>0.84</td>
<td>0.53</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>1.58</td>
<td>0.52</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td>1.46</td>
<td>0.55</td>
<td>0.54</td>
</tr>
<tr>
<td></td>
<td>1.74</td>
<td>0.75</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td>1.74</td>
<td>0.61</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td>1.05</td>
<td>0.62</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>1.05</td>
<td>0.71</td>
<td>1.28</td>
</tr>
<tr>
<td></td>
<td>1.29</td>
<td>0.50</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td>2.22</td>
<td>0.60</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>0.90</td>
<td>0.65</td>
<td>1.42</td>
</tr>
<tr>
<td></td>
<td>1.70</td>
<td>0.86</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td>0.81</td>
<td>1.07</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>6.66</td>
<td>2.35</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>0.93</td>
<td>5.18</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>0.34</td>
<td>1.24</td>
<td>0.51</td>
</tr>
<tr>
<td></td>
<td>0.45</td>
<td>1.28</td>
<td>0.46</td>
</tr>
<tr>
<td></td>
<td>1.85</td>
<td>4.38</td>
<td>1.45</td>
</tr>
<tr>
<td></td>
<td>1.32</td>
<td>3.15</td>
<td>1.03</td>
</tr>
</tbody>
</table>
## Finding CPU Bottlenecks

<table>
<thead>
<tr>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.50</td>
<td>3.16</td>
<td>0.98</td>
</tr>
<tr>
<td>1.42</td>
<td>3.00</td>
<td>1.00</td>
</tr>
<tr>
<td>0.90</td>
<td>2.94</td>
<td>1.00</td>
</tr>
<tr>
<td>0.69</td>
<td>2.36</td>
<td>1.15</td>
</tr>
<tr>
<td>1.08</td>
<td>2.72</td>
<td>0.86</td>
</tr>
<tr>
<td>1.49</td>
<td>4.22</td>
<td>1.49</td>
</tr>
<tr>
<td>0.97</td>
<td>2.91</td>
<td>0.91</td>
</tr>
<tr>
<td>0.89</td>
<td>3.05</td>
<td>0.90</td>
</tr>
<tr>
<td>1.36</td>
<td>3.02</td>
<td>1.07</td>
</tr>
<tr>
<td>1.12</td>
<td>2.95</td>
<td>0.95</td>
</tr>
<tr>
<td>1.63</td>
<td>3.47</td>
<td>1.02</td>
</tr>
<tr>
<td>0.96</td>
<td>2.95</td>
<td>0.98</td>
</tr>
<tr>
<td>2.75</td>
<td>5.55</td>
<td>1.83</td>
</tr>
<tr>
<td>2.11</td>
<td>1.47</td>
<td>0.51</td>
</tr>
<tr>
<td>0.44</td>
<td>1.50</td>
<td>0.48</td>
</tr>
<tr>
<td>0.67</td>
<td>1.46</td>
<td>0.51</td>
</tr>
<tr>
<td>2.07</td>
<td>3.93</td>
<td>3.13</td>
</tr>
<tr>
<td>0.71</td>
<td>4.36</td>
<td>1.93</td>
</tr>
<tr>
<td>1.75</td>
<td>3.31</td>
<td>1.15</td>
</tr>
<tr>
<td>2.39</td>
<td>1.79</td>
<td>1.02</td>
</tr>
<tr>
<td>0.96</td>
<td>1.71</td>
<td>0.81</td>
</tr>
<tr>
<td>0.57</td>
<td>1.70</td>
<td>0.73</td>
</tr>
<tr>
<td>1.88</td>
<td>1.81</td>
<td>0.58</td>
</tr>
<tr>
<td>0.59</td>
<td>1.72</td>
<td>0.55</td>
</tr>
<tr>
<td>2.28</td>
<td>3.74</td>
<td>1.72</td>
</tr>
<tr>
<td>2.66</td>
<td>0.84</td>
<td>0.70</td>
</tr>
<tr>
<td>0.64</td>
<td>0.82</td>
<td>0.64</td>
</tr>
<tr>
<td>0.30</td>
<td>0.80</td>
<td>0.62</td>
</tr>
<tr>
<td>1.78</td>
<td>0.70</td>
<td>0.63</td>
</tr>
<tr>
<td>7.20</td>
<td>2.35</td>
<td>1.04</td>
</tr>
<tr>
<td>0.49</td>
<td>0.21</td>
<td>0.50</td>
</tr>
<tr>
<td>9.99</td>
<td>0.26</td>
<td>0.54</td>
</tr>
<tr>
<td>4.28</td>
<td>0.23</td>
<td>0.66</td>
</tr>
<tr>
<td>0.04</td>
<td>0.26</td>
<td>1.94</td>
</tr>
<tr>
<td>3.55</td>
<td>0.52</td>
<td>0.66</td>
</tr>
<tr>
<td>4.56</td>
<td>0.59</td>
<td>0.62</td>
</tr>
<tr>
<td>5.38</td>
<td>0.33</td>
<td>0.68</td>
</tr>
<tr>
<td>4.44</td>
<td>0.33</td>
<td>0.65</td>
</tr>
<tr>
<td>4.35</td>
<td>0.30</td>
<td>0.73</td>
</tr>
<tr>
<td>3.76</td>
<td>0.27</td>
<td>0.60</td>
</tr>
<tr>
<td>3.72</td>
<td>0.30</td>
<td>0.64</td>
</tr>
<tr>
<td>3.75</td>
<td>0.26</td>
<td>0.58</td>
</tr>
<tr>
<td>4.79</td>
<td>0.33</td>
<td>0.75</td>
</tr>
<tr>
<td>4.68</td>
<td>0.33</td>
<td>0.85</td>
</tr>
<tr>
<td>3.00</td>
<td>0.22</td>
<td>0.53</td>
</tr>
<tr>
<td>2.44</td>
<td>0.26</td>
<td>0.83</td>
</tr>
<tr>
<td>14.87</td>
<td>0.69</td>
<td>1.59</td>
</tr>
<tr>
<td>8.68</td>
<td>0.96</td>
<td>1.96</td>
</tr>
<tr>
<td>3.44</td>
<td>0.47</td>
<td>0.96</td>
</tr>
<tr>
<td>3.73</td>
<td>0.22</td>
<td>0.65</td>
</tr>
<tr>
<td>3.06</td>
<td>0.72</td>
<td>0.65</td>
</tr>
<tr>
<td>3.86</td>
<td>0.35</td>
<td>1.13</td>
</tr>
<tr>
<td>3.32</td>
<td>0.26</td>
<td>0.57</td>
</tr>
<tr>
<td>3.21</td>
<td>0.26</td>
<td>0.62</td>
</tr>
</tbody>
</table>
### Finding CPU Bottlenecks

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.84</td>
<td>0.26</td>
<td>0.60</td>
</tr>
<tr>
<td>4.85</td>
<td>0.33</td>
<td>0.72</td>
</tr>
<tr>
<td>4.16</td>
<td>0.32</td>
<td>0.70</td>
</tr>
<tr>
<td>3.96</td>
<td>0.30</td>
<td>0.69</td>
</tr>
<tr>
<td>2.60</td>
<td>0.82</td>
<td>0.66</td>
</tr>
<tr>
<td>8.72</td>
<td>0.47</td>
<td>0.69</td>
</tr>
<tr>
<td>0.49</td>
<td>0.31</td>
<td>1.50</td>
</tr>
<tr>
<td>0.46</td>
<td>0.28</td>
<td>0.77</td>
</tr>
<tr>
<td>7.54</td>
<td>3.66</td>
<td>0.90</td>
</tr>
<tr>
<td>7.50</td>
<td>0.27</td>
<td>0.71</td>
</tr>
<tr>
<td>0.06</td>
<td>0.32</td>
<td>2.37</td>
</tr>
<tr>
<td>6.07</td>
<td>0.28</td>
<td>0.97</td>
</tr>
<tr>
<td>3.68</td>
<td>0.27</td>
<td>0.52</td>
</tr>
<tr>
<td>6.39</td>
<td>5.86</td>
<td>4.48</td>
</tr>
<tr>
<td>4.66</td>
<td>0.29</td>
<td>1.28</td>
</tr>
<tr>
<td>0.05</td>
<td>0.26</td>
<td>11.86</td>
</tr>
<tr>
<td>8.87</td>
<td>12.64</td>
<td>1.25</td>
</tr>
<tr>
<td>3.32</td>
<td>0.26</td>
<td>0.58</td>
</tr>
<tr>
<td>6.22</td>
<td>4.77</td>
<td>1.26</td>
</tr>
<tr>
<td>3.49</td>
<td>0.31</td>
<td>0.86</td>
</tr>
<tr>
<td>11.32</td>
<td>10.49</td>
<td>1.26</td>
</tr>
<tr>
<td>10.27</td>
<td>15.09</td>
<td>1.78</td>
</tr>
<tr>
<td>12.50</td>
<td>1.34</td>
<td>2.53</td>
</tr>
<tr>
<td>7.66</td>
<td>4.74</td>
<td>0.58</td>
</tr>
<tr>
<td>0.03</td>
<td>0.24</td>
<td>0.32</td>
</tr>
<tr>
<td>4.43</td>
<td>0.30</td>
<td>0.56</td>
</tr>
<tr>
<td>9.75</td>
<td>2.94</td>
<td>1.68</td>
</tr>
<tr>
<td>17.93</td>
<td>0.47</td>
<td>0.56</td>
</tr>
<tr>
<td>3.81</td>
<td>0.35</td>
<td>1.04</td>
</tr>
<tr>
<td>0.20</td>
<td>2.84</td>
<td>2.72</td>
</tr>
<tr>
<td>10.06</td>
<td>0.28</td>
<td>0.92</td>
</tr>
<tr>
<td>5.74</td>
<td>0.72</td>
<td>1.92</td>
</tr>
<tr>
<td>0.07</td>
<td>0.87</td>
<td>0.53</td>
</tr>
<tr>
<td>2.05</td>
<td>0.95</td>
<td>2.03</td>
</tr>
</tbody>
</table>

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.
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 604: gfxinfo Output, In Stacked Column Chart](image-url)
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:
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](https://commonsware.com).

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`).
The syntax changed for Android 4.3 and higher to simplify matters, combining the two `systrace.py` commands into one:

```bash
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 606: Systrace Toolbar Button in Devices View](image)

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`: 
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
Finding CPU Bottlenecks

- `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:
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:
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:
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.
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.
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.
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.
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
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 aptitude install make gawk`, or use the Synaptic Package Manager, 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 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).
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:

```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,
...
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]; }
}
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_WeakBenchNsievenative
- Java_com_commonsware_abj_weakbench_WeakBenchSpecnative

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:

```makefile
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).
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.
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.
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:

```java
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:

```java
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) {
```
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)findViewByld(R.id.nsieve_java);

        result.setText("running...");
    }

    @Override
    protected Void doInBackground(Void... unused) {
        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;
}
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();
    }
}
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);
    }
}
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.
Focus On: NDK

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.
FOCUS ON: NDK

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, the NDK, and Architecture Flavors

The new Gradle for Android-based build system also can help here, by making it easier for you to create dedicated versions of your APK with just the .so files for a single architecture. If you distribute solely through the Play Store, you can upload each of those APK files to a single Play Store listing, with the Play Store distributing the proper APK to the device based on its CPU. This way, you get NDK speed with less bloat.

The chapter on advanced Gradle for Android techniques has a section covering the use of Gradle for Android for NDK projects.
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.
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
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.
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 bind 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.
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](https://commonsware.com/CPU-Java) 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...
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
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
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
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.
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
<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
        android:layout_width="match_parent"
        android:id="@+id/button2"
        android:layout_height="wrap_content"
        android:text="@string/async"
        android:onClick="sendAsync"/>
</LinearLayout>
```

The activity itself simply has sendAsync() and sendNonAsync() methods, each invoking sendBroadcast() to a different BroadcastReceiver implementation:

```java
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:

```java
package com.commonsware.android.tuning.goasync;

import android.content.BroadcastReceiver;

public class NonAsyncReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Simulate time-consuming work
    }
}
```
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.

2339
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:

```java
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));
    }
}
```
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():

```java
package com.commonsware.android.tuningprefs;
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();
  }
}

public static final AbstractPrefsPersistStrategy INSTANCE=
  initImpl();

On Honeycomb and higher, it uses a separate strategy class that uses the new apply() method:

```java
package com.commonsware.android.tuningprefs;
import android.content.SharedPreferences.Editor;

public class ApplyStrategy extends AbstractPrefsPersistStrategy {
  @Override
  void persistAsync(Editor 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.
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.
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.
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 StackOverflow question with the solution presented in this book offered as an answer.
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:
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 `ListView`s, 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...
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:
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.
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:

![Three-PANE Demo, As Initially Launched, Showing Overdraw](image)

*Figure 613: Three-PANE Demo, As Initially Launched, Showing Overdraw*
FINDING AND ELIMINATING JANK

Figure 614: Three-Pane Demo, Left and Middle Panes, Showing Overdraw

Figure 615: Three-Pane Demo, Middle and Right Panes, Showing Overdraw
On the other hand, bringing up the Contacts app on the same Nexus 7 shows significantly more 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.
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.

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.
Figure 6.19: 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:
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](https://commonsware.com) to a StackOverflow question about this three-
Finding and Eliminating Jank

pane animation, asked by the person who commented about the dropped frames on the original StackOverflow 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](https://commonsware.com) sample app.

Now, our `showLeft()` and `hideLeft()` methods immediately change the width of the middle pane, rather than arranging its animation:

```java
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.setLayoutParams().width=value;
    requestLayout();
}
```
This does not provide nearly as good of a UI as the original. 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.

What we can say is that the revised solution does reduce the jank, as seen in this gfxinfo output:
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.
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
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.
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.
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.
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.
Focus on: TrafficStats

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:

```java
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) {
```

2365
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:

```java
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
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
<?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:textStyle="bold"
            android:textSize="20sp"/>
        <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"/>
    </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:

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

2368
**setContentView(R.layout.main);**

```java
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:

```java
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);
    }
```
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):

```java
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.
Using TrafficMonitor

Running the activity gives you the initial received and sent counts (in bytes):

![TrafficMonitor Activity](image)

*Figure 621: The TrafficMonitor sample application, as initially launched*

Tapping Take Snapshot grabs a second snapshot and compares the two:
FOCUS ON: TRAFFIC STATS

Figure 622: 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.browser=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)
```
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.
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.
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
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...
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:
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
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:
MEASURING BANDWIDTH CONSUMPTION

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:

---

Figure 624: Network Statistics View
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.
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.
BEING SMARTER ABOUT BANDWIDTH

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:

```java
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:
BEING SMARTER ABOUT BANDWIDTH

// 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
On subsequent requests, you can add the If-None-Match header and handle both cases:

```java
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.
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
BEING SMARTER ABOUT BANDWIDTH

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
BEING SMARTER ABOUT BANDWIDTH

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.
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.
Connectivity

If the user might not care how much bandwidth you consume, so long as it is un-metered 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.
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
current data connection may involve bandwidth charges. You can examine this value and steer your bandwidth consumption accordingly.
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.
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.
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
reusing these pre-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)
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 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.
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.
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 downscaled 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.

**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.

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](https://commonsware.com) is down for maintenance).

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.
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:
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

Some developers 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. With MAT, you have your choice.

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 DDMS perspective or the standalone DDMS utility.

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:
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.

**DDMS Perspective**

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.

**Standalone DDMS**

Once you click the toolbar button for the heap dump, DDMS will create the dump for you, in a file chosen by you via your platform's standard file-save dialog.

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.

**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 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:

```java
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();
        }
    }
}
```

---

2408
Finding Memory Leaks with MAT

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"
    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
    the repository. -->
    <!-- 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
    the repository. -->
</project>

2409
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 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"
/>
Finding Memory Leaks with MAT

<!-- 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: 
  - 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:

![Figure 627: The MAT Eclipse perspective, as initially opened](image)

If you used standalone DDMS 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 will 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:
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:
The histogram initially displays the top culprits in terms of “shallow heap” — the amount of memory those objects hold onto directly:

```
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Objects</th>
<th>Shallow-Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>10,830</td>
<td>947,608</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>12,140</td>
<td>291,360</td>
</tr>
<tr>
<td>byte[]</td>
<td>1,349</td>
<td>217,848</td>
</tr>
<tr>
<td>java.util.HashMap$Map$MapEntry</td>
<td>3,374</td>
<td>66,576</td>
</tr>
<tr>
<td>java.lang.Class</td>
<td>2,169</td>
<td>52,560</td>
</tr>
<tr>
<td>int[]</td>
<td>722</td>
<td>56,600</td>
</tr>
<tr>
<td>java.lang.String[]</td>
<td>798</td>
<td>46,084</td>
</tr>
<tr>
<td>org.bouncycastle.asn1.DERSequence</td>
<td>1,246</td>
<td>39,672</td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>2,446</td>
<td>39,136</td>
</tr>
<tr>
<td>java.lang.Object[]</td>
<td>495</td>
<td>32,976</td>
</tr>
<tr>
<td>java.util.HashMap$Map$MapEntry</td>
<td>30</td>
<td>29,713</td>
</tr>
<tr>
<td>java.util.HashMap$Map$MapEntry</td>
<td>704</td>
<td>16,896</td>
</tr>
<tr>
<td>org.bouncycastle.asn1.DERObjectIdentifier</td>
<td>1,035</td>
<td>16,560</td>
</tr>
<tr>
<td>org.apache.harmony.luni.util.twkeyHash$Map$Entry</td>
<td>405</td>
<td>14,660</td>
</tr>
<tr>
<td>java.util.ArrayList</td>
<td>404</td>
<td>12,928</td>
</tr>
<tr>
<td>org.apache.harmony.security.x509.AttributeValue</td>
<td>435</td>
<td>10,440</td>
</tr>
<tr>
<td>org.apache.harmony.security.x509.$Map$Entry</td>
<td>234</td>
<td>9,360</td>
</tr>
<tr>
<td>java.security.Provider$Service</td>
<td>8</td>
<td>7,640</td>
</tr>
<tr>
<td>org.bouncycastle.asn1.DERPrintableString</td>
<td>182</td>
<td>7,280</td>
</tr>
<tr>
<td>java.util.HashMap$Map$MapEntry</td>
<td>448</td>
<td>7,104</td>
</tr>
<tr>
<td>java.util.HashMap$Map$MapEntry</td>
<td>70</td>
<td>6,718</td>
</tr>
<tr>
<td>long[]</td>
<td>127</td>
<td>6,664</td>
</tr>
<tr>
<td>org.bouncycastle.asn1.x509.$Map$Name$Element$List</td>
<td>112</td>
<td>6,272</td>
</tr>
<tr>
<td>org.xml*cosan.tagmap$Element$Type</td>
<td>109</td>
<td>5,785</td>
</tr>
<tr>
<td>Total: 24 of 2,143 entries</td>
<td>46,502</td>
<td>1,737,360</td>
</tr>
</tbody>
</table>
```

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:
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:
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 632:** A filtered histogram, showing com.commonsware.* objects

---

**Figure 633:** The GC roots holding onto an activity
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:
This will open up another tab in the same tool, showing “the biggest objects” by retained heap:

![Figure 635: The MAT Dominator Tree tab](image)

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.
Widget in Static Data Member

The screen shots from above are mostly taken from the MAT/StaticWidget sample project, where we do something naughty:

```java
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:

```java
package com.commonsware.android.tuning.mat;

import android.app.Activity;
import android.os.Bundle;
import android.os.SystemClock;
```
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:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Objects</th>
<th>Shallow Heap</th>
<th>Retained Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.commonsware.android.tuning.mat.LeakedThreadActivity</td>
<td>1</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>com.commonsware.android.tuning.mat.LeakedThreadActivity$1</td>
<td>1</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>Total: 2 entries</td>
<td>2</td>
<td>232</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 636: 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:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Ref. Objects</th>
<th>Shallow Heap</th>
<th>Ref. Shallow Heap</th>
<th>Retained Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.commonsware.android.tuning.mat.LeakedThreadActivity$1 @ 0x44e3ae0</td>
<td>1</td>
<td>152</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td>com.commonsware.android.tuning.mat.LeakedThreadActivity@0x44e3158</td>
<td>1</td>
<td>152</td>
<td>152</td>
<td>2,320</td>
</tr>
</tbody>
</table>

*Figure 637: 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:
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 638: The com.commonsware objects in the RandomAppOfCrap heap](image)

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
outgoing references (what these objects point to). In this case, showing incoming references will bring up the following:

![Figure 639: The incoming references for one WeatherDemo instance](image)

**Figure 639: The incoming references for one 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:
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
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.
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
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.
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)
Tapping on that entry brings up a screen that describes the current state of the system, with respect to RAM:
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.
The Roster

The list beneath the summary shows some running processes. What is included in this list depends on what mode Process Stats is 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:
Tapping that gives you a roster of available timeframes:

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
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:
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.
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)
**Issues with System RAM**

* 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.rememberthemailk.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%
ISSUES WITH SYSTEM RAM

(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)

2438
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., `-csvProc`) 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
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:

<table>
<thead>
<tr>
<th>Applications Memory Usage (kB):</th>
<th>Uptime: 95955008 Realtime: 788076654</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total PSS by process:</strong></td>
<td></td>
</tr>
<tr>
<td>120505 kB: com.google.android.apps.maps (pid 17490 / activities)</td>
<td></td>
</tr>
<tr>
<td>74627 kB: com.android.launcher (pid 1696 / activities)</td>
<td></td>
</tr>
<tr>
<td>62422 kB: system (pid 1366)</td>
<td></td>
</tr>
<tr>
<td>57757 kB: surfaceflinger (pid 968)</td>
<td></td>
</tr>
<tr>
<td>51706 kB: com.fsck.k9 (pid 2937 / activities)</td>
<td></td>
</tr>
<tr>
<td>44725 kB: com.tripit (pid 2414 / activities)</td>
<td></td>
</tr>
<tr>
<td>41642 kB: com.android.systemui (pid 1498 / activities)</td>
<td></td>
</tr>
<tr>
<td>38546 kB: com.google.android.inputmethod.latin (pid 1635)</td>
<td></td>
</tr>
<tr>
<td>36640 kB: com.rememberthemilk.MobileRTM (pid 12255 / activities)</td>
<td></td>
</tr>
<tr>
<td>35518 kB: com.szyk.myheart (pid 24618 / activities)</td>
<td></td>
</tr>
<tr>
<td>32588 kB: com.stackexchange.marvin (pid 28230 / activities)</td>
<td></td>
</tr>
<tr>
<td>27128 kB: com.android.phone (pid 1667)</td>
<td></td>
</tr>
<tr>
<td>23641 kB: com.android.mms (pid 15197 / activities)</td>
<td></td>
</tr>
<tr>
<td>23236 kB: nz.co.softwarex.hundredpushupsfree (pid 20599 / activities)</td>
<td></td>
</tr>
<tr>
<td>22483 kB: org.wikipedia (pid 11895 / activities)</td>
<td></td>
</tr>
<tr>
<td>18044 kB: com.commonsware.books.android (pid 12036 / activities)</td>
<td></td>
</tr>
<tr>
<td>16968 kB: com.google.process.location (pid 1775)</td>
<td></td>
</tr>
<tr>
<td>16895 kB: com.google.android.gms (pid 1651)</td>
<td></td>
</tr>
<tr>
<td>16521 kB: com.google.process.gapps (pid 1803)</td>
<td></td>
</tr>
<tr>
<td>15822 kB: com.evernote (pid 26284)</td>
<td></td>
</tr>
<tr>
<td>15219 kB: android.process.acore (pid 11926)</td>
<td></td>
</tr>
<tr>
<td>11336 kB: zygote (pid 969)</td>
<td></td>
</tr>
<tr>
<td>10694 kB: com.csipsimple: sipStack (pid 25539)</td>
<td></td>
</tr>
<tr>
<td>9575 kB: com.google.android.youtube (pid 31932)</td>
<td></td>
</tr>
<tr>
<td>8390 kB: tunein.player.pro (pid 2699)</td>
<td></td>
</tr>
<tr>
<td>7860 kB: com.android.bluetooth (pid 5513)</td>
<td></td>
</tr>
<tr>
<td>7669 kB: org.mozilla.firefox (pid 20839)</td>
<td></td>
</tr>
<tr>
<td>6786 kB: com.espn.radio:com.urbanairship.process (pid 2526)</td>
<td></td>
</tr>
<tr>
<td>6719 kB: mediaserver (pid 971)</td>
<td></td>
</tr>
<tr>
<td>6599 kB: com.google.android.gallery3d (pid 31894)</td>
<td></td>
</tr>
<tr>
<td>6493 kB: com.android.nfc (pid 1681)</td>
<td></td>
</tr>
<tr>
<td>5916 kB: com.google.android.apps.genie.geniewidget (pid 22781)</td>
<td></td>
</tr>
<tr>
<td>5677 kB: android.process.media (pid 25240)</td>
<td></td>
</tr>
<tr>
<td>4308 kB: com.csipsimple (pid 28166)</td>
<td></td>
</tr>
<tr>
<td>4145 kB: com.google.android.deskclock (pid 12379)</td>
<td></td>
</tr>
</tbody>
</table>
## Issues with System RAM

<table>
<thead>
<tr>
<th>Process Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.guywmustang.silentwidget (pid 14616)</td>
<td>3472 kB</td>
</tr>
<tr>
<td>rild (pid 967)</td>
<td>3349 kB</td>
</tr>
<tr>
<td>drmserver (pid 970)</td>
<td>2447 kB</td>
</tr>
<tr>
<td>ks (pid 585)</td>
<td>1972 kB</td>
</tr>
<tr>
<td>netd (pid 965)</td>
<td>1876 kB</td>
</tr>
<tr>
<td>wpa_supplicant (pid 26091)</td>
<td>1282 kB</td>
</tr>
<tr>
<td>mm-qcamera-daemon (pid 982)</td>
<td>1217 kB</td>
</tr>
<tr>
<td>sdcard (pid 981)</td>
<td>1116 kB</td>
</tr>
<tr>
<td>sensors.qcom (pid 979)</td>
<td>618 kB</td>
</tr>
<tr>
<td>netmgrd (pid 976)</td>
<td>577 kB</td>
</tr>
<tr>
<td>void (pid 163)</td>
<td>500 kB</td>
</tr>
<tr>
<td>bridgemgrd (pid 974)</td>
<td>486 kB</td>
</tr>
<tr>
<td>thermald (pid 977)</td>
<td>476 kB</td>
</tr>
<tr>
<td>keystore (pid 973)</td>
<td>462 kB</td>
</tr>
<tr>
<td>/init (pid 1)</td>
<td>439 kB</td>
</tr>
<tr>
<td>qmuxd (pid 975)</td>
<td>375 kB</td>
</tr>
<tr>
<td>ueventd (pid 139)</td>
<td>262 kB</td>
</tr>
<tr>
<td>dhcpcd (pid 15630)</td>
<td>230 kB</td>
</tr>
<tr>
<td>qseecomd (pid 1022)</td>
<td>214 kB</td>
</tr>
<tr>
<td>adbd (pid 961)</td>
<td>210 kB</td>
</tr>
<tr>
<td>installld (pid 972)</td>
<td>189 kB</td>
</tr>
<tr>
<td>mpdecision (pid 978)</td>
<td>181 kB</td>
</tr>
<tr>
<td>rmt_storage (pid 164)</td>
<td>176 kB</td>
</tr>
<tr>
<td>dumpsys (pid 489)</td>
<td>169 kB</td>
</tr>
<tr>
<td>qcks (pid 165)</td>
<td>149 kB</td>
</tr>
<tr>
<td>debuggerd (pid 966)</td>
<td>140 kB</td>
</tr>
<tr>
<td>healthd (pid 161)</td>
<td>135 kB</td>
</tr>
<tr>
<td>efsks (pid 569)</td>
<td>115 kB</td>
</tr>
<tr>
<td>servicemanager (pid 162)</td>
<td>111 kB</td>
</tr>
<tr>
<td>qseecomd (pid 986)</td>
<td>111 kB</td>
</tr>
</tbody>
</table>

**Total PSS by OOM adjustment:**

<table>
<thead>
<tr>
<th>Process Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>surfaceflinger (pid 968)</td>
<td>57757 kB</td>
</tr>
<tr>
<td>zygote (pid 969)</td>
<td>11336 kB</td>
</tr>
<tr>
<td>mediasever (pid 971)</td>
<td>6719 kB</td>
</tr>
<tr>
<td>rild (pid 967)</td>
<td>3349 kB</td>
</tr>
<tr>
<td>drmserver (pid 970)</td>
<td>2447 kB</td>
</tr>
<tr>
<td>ks (pid 585)</td>
<td>1972 kB</td>
</tr>
<tr>
<td>netd (pid 965)</td>
<td>1876 kB</td>
</tr>
<tr>
<td>wpa_supplicant (pid 26091)</td>
<td>1282 kB</td>
</tr>
<tr>
<td>mm-qcamera-daemon (pid 982)</td>
<td>1217 kB</td>
</tr>
<tr>
<td>sdcard (pid 981)</td>
<td>1116 kB</td>
</tr>
<tr>
<td>sensors.qcom (pid 979)</td>
<td>618 kB</td>
</tr>
<tr>
<td>netmgrd (pid 976)</td>
<td>577 kB</td>
</tr>
<tr>
<td>void (pid 163)</td>
<td>500 kB</td>
</tr>
<tr>
<td>bridgemgrd (pid 974)</td>
<td>486 kB</td>
</tr>
<tr>
<td>thermald (pid 977)</td>
<td>476 kB</td>
</tr>
<tr>
<td>keystore (pid 973)</td>
<td>462 kB</td>
</tr>
<tr>
<td>/init (pid 1)</td>
<td>439 kB</td>
</tr>
<tr>
<td>qmuxd (pid 975)</td>
<td>375 kB</td>
</tr>
<tr>
<td>ueventd (pid 139)</td>
<td>262 kB</td>
</tr>
<tr>
<td>dhcpcd (pid 15630)</td>
<td>230 kB</td>
</tr>
<tr>
<td>qseecomd (pid 1022)</td>
<td>214 kB</td>
</tr>
</tbody>
</table>

2441
## Issues with System RAM

<table>
<thead>
<tr>
<th>Memory</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>212 kB</td>
<td>adb (pid 961)</td>
</tr>
<tr>
<td>210 kB</td>
<td>installld (pid 972)</td>
</tr>
<tr>
<td>189 kB</td>
<td>mpdecision (pid 978)</td>
</tr>
<tr>
<td>181 kB</td>
<td>rmt_storage (pid 164)</td>
</tr>
<tr>
<td>176 kB</td>
<td>dumpsys (pid 489)</td>
</tr>
<tr>
<td>169 kB</td>
<td>qcks (pid 165)</td>
</tr>
<tr>
<td>149 kB</td>
<td>debuggerd (pid 966)</td>
</tr>
<tr>
<td>140 kB</td>
<td>healthd (pid 161)</td>
</tr>
<tr>
<td>135 kB</td>
<td>efsks (pid 569)</td>
</tr>
<tr>
<td>115 kB</td>
<td>servicemanager (pid 162)</td>
</tr>
<tr>
<td>111 kB</td>
<td>qseecomd (pid 986)</td>
</tr>
<tr>
<td>62422 kB</td>
<td>System</td>
</tr>
<tr>
<td>75263 kB</td>
<td>Persistent</td>
</tr>
<tr>
<td>41642 kB</td>
<td>com.android.systemui (pid 1498 / activities)</td>
</tr>
<tr>
<td>27128 kB</td>
<td>com.android.phone (pid 1667)</td>
</tr>
<tr>
<td>6493 kB</td>
<td>com.android.nfc (pid 1681)</td>
</tr>
<tr>
<td>74627 kB</td>
<td>Foreground</td>
</tr>
<tr>
<td>74627 kB</td>
<td>com.android.launcher (pid 1696 / activities)</td>
</tr>
<tr>
<td>63374 kB</td>
<td>Visible</td>
</tr>
<tr>
<td>38546 kB</td>
<td>com.google.android.inputmethod.latin (pid 1635)</td>
</tr>
<tr>
<td>16968 kB</td>
<td>com.google.process.location (pid 1775)</td>
</tr>
<tr>
<td>7860 kB</td>
<td>com.android.bluetooth (pid 5513)</td>
</tr>
<tr>
<td>10694 kB</td>
<td>Perceptible</td>
</tr>
<tr>
<td>10694 kB</td>
<td>com.csipsimple:sipStack (pid 25539)</td>
</tr>
<tr>
<td>5677 kB</td>
<td>A Services</td>
</tr>
<tr>
<td>5677 kB</td>
<td>android.process.media (pid 25240)</td>
</tr>
<tr>
<td>51706 kB</td>
<td>Previous</td>
</tr>
<tr>
<td>51706 kB</td>
<td>com.fsck.k9 (pid 2937 / activities)</td>
</tr>
<tr>
<td>76422 kB</td>
<td>B Services</td>
</tr>
<tr>
<td>44725 kB</td>
<td>com.tripit (pid 2414 / activities)</td>
</tr>
<tr>
<td>16521 kB</td>
<td>com.google.process.gapps (pid 1803)</td>
</tr>
<tr>
<td>8390 kB</td>
<td>tunein.player.pro (pid 2699)</td>
</tr>
<tr>
<td>6786 kB</td>
<td>com.espn.radio:com.urbanairship.process (pid 2526)</td>
</tr>
<tr>
<td>402275 kB</td>
<td>Cached</td>
</tr>
<tr>
<td>120505 kB</td>
<td>com.google.android.apps.maps (pid 17490 / activities)</td>
</tr>
<tr>
<td>36640 kB</td>
<td>com.rememberthemilk.MobileRTM (pid 12255 / activities)</td>
</tr>
<tr>
<td>35518 kB</td>
<td>com.szyk.myheart (pid 24618 / activities)</td>
</tr>
<tr>
<td>32588 kB</td>
<td>com.stackexchange.marvin (pid 28230 / activities)</td>
</tr>
<tr>
<td>23641 kB</td>
<td>com.android.mms (pid 15197 / activities)</td>
</tr>
<tr>
<td>23236 kB</td>
<td>nz.co.softwarex.hundredpushupsfree (pid 20599 / activities)</td>
</tr>
<tr>
<td>22483 kB</td>
<td>org.wikipedia (pid 11895 / activities)</td>
</tr>
<tr>
<td>18044 kB</td>
<td>com.commonscode.books.android (pid 12036 / activities)</td>
</tr>
<tr>
<td>16895 kB</td>
<td>com.google.android.gms (pid 1651)</td>
</tr>
<tr>
<td>15822 kB</td>
<td>com.evernote (pid 26284)</td>
</tr>
<tr>
<td>15219 kB</td>
<td>android.process.acore (pid 11926)</td>
</tr>
<tr>
<td>9575 kB</td>
<td>com.google.android.youtube (pid 31932)</td>
</tr>
<tr>
<td>7669 kB</td>
<td>org.mozilla.firefox (pid 20839)</td>
</tr>
<tr>
<td>6599 kB</td>
<td>com.google.android.gallery3d (pid 31894)</td>
</tr>
<tr>
<td>5916 kB</td>
<td>com.google.android.apps.genie.geniewidget (pid 22781)</td>
</tr>
<tr>
<td>4308 kB</td>
<td>com.csipsimple (pid 28166)</td>
</tr>
<tr>
<td>4145 kB</td>
<td>com.google.android.deskclock (pid 12379)</td>
</tr>
<tr>
<td>3472 kB</td>
<td>com.guywmustang.silentwidget (pid 14616)</td>
</tr>
</tbody>
</table>
ISSUES WITH SYSTEM RAM

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

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

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: 9612083 Realtime: 788242449

** MEMINFO in pid 12036 [com.commonsware.books.android] **
Pss          Private          Private          Swapped          Heap          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       4         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

2443
ISSUES WITH SYSTEM RAM

<table>
<thead>
<tr>
<th>Objects</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Views</td>
<td>48</td>
<td>ViewRootImpl:</td>
</tr>
<tr>
<td>AppContexts</td>
<td>3</td>
<td>Activities:</td>
</tr>
<tr>
<td>Assets</td>
<td>2</td>
<td>AssetManagers:</td>
</tr>
<tr>
<td>Local Binders</td>
<td>8</td>
<td>Proxy Binders:</td>
</tr>
<tr>
<td>Death Recipients</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>OpenSSL Sockets</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_USED</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>PAGECACHE_OVERFLOW</td>
<td>3</td>
<td>MALLOC_SIZE:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATABASES</th>
<th>pgpsz</th>
<th>dbsz</th>
<th>Lookaside(b)</th>
<th>cache</th>
<th>Dbname</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>6263</td>
<td>17</td>
<td>0/16/1</td>
<td>/data/data/com.commonsware.books.android/databases/booksearch.db</td>
</tr>
</tbody>
</table>

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:
ISSUES WITH SYSTEM RAM

- The total memory on the device (totalMem)
- 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 (lrup), 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 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.
Most Android devices are powered by batteries — Google 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.
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”.

![Battery Screen](image)

*Figure 647: 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.
ISSUES WITH BATTERY LIFE

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.
This part of the book will help you identify when the battery is low and strategies for making use of that information.
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.
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:

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.

2452
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:
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:
Figure 649: 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:
Figure 650: 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:
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:
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:
POWER MEASUREMENT OPTIONS

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
```
### Power Measurement Options

- 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 "msmb_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 wakelock 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

2460
and per-process information (here, showing power used by PowerTutor itself):

```markdown
#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.
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
• 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.
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
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:
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.
Sources of Power Drain

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,
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
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.
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
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.
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.
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.
ADDRESSING APPLICATION SIZE ISSUES

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 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

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.
ADDRESSING APPLICATION SIZE ISSUES

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 `runProguard true` and a custom ProGuard configuration file:

```java
android {
    buildTypes {
        debug {
            runProguard 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.

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 interoperate, 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.

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, Grable for Android makes this fairly straight-forward, using product flavors for the CPU architectures:

```java
productFlavors {
  x86 {
    ndk {
      abiFilter "x86"
    }
  }
  arm {
    ndk {
      abiFilter "armeabi-v7a"
    }
  }
  mips {
    ndk {
      abiFilter "mips"
    }
  }
}
```

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.

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-CA and fr-CA. 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.
ADDRESSING APPLICATION SIZE ISSUES

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
<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, 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.
Trail: Alternatives for App Development
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.
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).
The Role of Alternative Environments

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.
THE ROLE OF ALTERNATIVE ENVIRONMENTS

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.
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](https://commonsware.com) would be a good idea, as would reading the [introduction to this trail](https://commonsware.com).

**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).
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:
Figure 654: 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:
You can “check off” individual items, which are then displayed in strike-through:
Figure 656: 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:
“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"
```
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
```
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.
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:

```javascript
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:

```javascript
/*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:
/*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;
},

HTML5

2496
**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](https://whatwg.org) working on this standard has set it aside for the time being.

You might consider evaluating [Lawnchair](https://github.com/lawnchair/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](https://webdriver.io) to [QUnit](https://qunitjs.com) to [Jasmine](https://jasmine.github.io).

For testing on Android proper — to ensure there are no issues related to Android’s browser implementation — you can use Selenium’s [Android Driver](https://github.com/seleniumhq/selenium-android-driver) or [Remote Control](https://seleniumhq.github.io/selenium/docs/api/java/index.html) modes.

**Signing and Distribution**

Unlike native Android applications, you do not need to worry about signing your HTML5 applications.
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.
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,
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.
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.
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.
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., jQuery 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
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:

![PhoneGap Example Application](image)

**Figure 658:** 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
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.
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 unZIPPped the PhoneGap ZIP file, copy the PhoneGap JAR file to the lib/ 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 unZIPPped 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"
When you want to test the application, you can build and install it like any other Android application (e.g., \texttt{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.

\textbf{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:

\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig659.png}
\caption{Creating your first project in PhoneGap/Build}
\end{figure}

You will be able to add new projects later on via a New App button, which gives you the same set of options.
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:

![Building your first project in PhoneGap/Build](image)

*Figure 660: 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:
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:
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.
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:

```java
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
<manifest version="1.0" encoding="utf-8">
  <application>
    <activity>
      <configChanges>orientation|keyboardHidden</configChanges>
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
      </intent-filter>
      <meta-data android:name="android.appператор.launcher" android:icon="/res/drawable-cw" android:label="/res/values/string/app_name"/>
      <meta-data android:name="android.appператор.title" android:label="/res/values/string/app_name"/>
    </activity>
  </application>
  <uses-permission android:name="android.permission.INTERNET"/>
</manifest>
```
PHONEGAP

```xml
<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.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:

```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" 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"
```
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()`)
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):

```javascript
// 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:
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.
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
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.
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.
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.
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 the Blackberry Playbook tablet, 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
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:
Other Alternative Environments

Figure 6.64: 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:
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
OTHER ALTERNATIVE ENVIRONMENTS

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.
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.
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:

```java
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*
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.
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.
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 something 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.
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.
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
ANTI-PATTERNS

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.

2536
ANTI-PATTERNS

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 StackOverflow’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.
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
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](https://medium.com/@RomainGuy))
- “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](https://medium.com/@DianneHackborn))
- “There is no reason or need to call `exit()`” ([Dianne Hackborn](https://medium.com/@DianneHackborn))
- “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](https://medium.com/@DianneHackborn))

**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**
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.
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
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.
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::
Trail: Widget Catalog
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 AdapterViewFlipper:

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.
A Sample Usage

The sample project can be found in WidgetCatalog/AdapterViewFlipper.

Layout:

```xml
<?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:

```java
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.
**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 2.1 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.
A Sample Usage

The sample project can be found in `WidgetCatalog/CalendarView`.

Layout:

```xml
<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:

```java
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();
    }
}
```
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 666: Android 4.0
Figure 667: Android 4.1
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.

**A Sample Usage**

The sample project can be found in WidgetCatalog/DatePicker.

Layout:
Activity:

```java
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) {
        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);
        }
    }

    // other code...
}
```
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) {
    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.
**Widget Catalog: DatePicker**

*Figure 668: Android 2.3.3*

*Figure 669: Android 4.0.3, with CalendarView, Portrait*
Figure 670: Android 4.0.3, without CalendarView, Portrait

Figure 671: Android 4.0.3, with CalendarView, Landscape
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 ExpandableListAdapter, 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
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`. 
### Widget Catalog: ExpandableListView

**Layout:**

```xml
<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:**

```json
{
    "Group B": ["Child B1", "Child B2"],
    "Group C": ["Child C1"],
    "Group D": [],
    "Group E": ["Child E1", "Child E2", "Child E3"]
}
```

**Activity:**

```java
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 {
    @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;
```

2559
StringBuffer buf = new StringBuffer();

try {
    while ((str = in.readLine()) != null) {
        buf.append(str);
        buf.append('
');
    }
    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();
}
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:
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(),
"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.
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 672: Android 2.3.3, 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.
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:
Activity:

```java
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);
    }
}
```
@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 674: Android 2.3.3
Figure 675: Android 4.1
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 View Pager 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.

**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.
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
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/drawer"
        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>
```
Activity:

```java
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 676: Android 2.3.3, with Drawer Closed](image-url)
Figure 677: Android 2.3.3, with Drawer Open

Figure 678: Android 4.0.3, with Drawer Closed
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.
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](https://commonsware.com).

Activity Layout:

```xml
<?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:

```xml
<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:

```java
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 {
    StackView stack;

    @Override
    public void onCreate(Bundle icicle) {
        ...
    }
```
super.onCreate(icicle);
setContentView(R.layout.main);

stack=(StackView)findViewByld(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=
        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:
Figure 679: Android 4.0.3, As Initially Seen
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
**Widget Catalog: TabHost and TabWidget**

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
<?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"
```
Activity:

```java
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);
    }
}
```
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 680: Android 2.3.3*
Figure 681: Android 4.0.3
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.
**Widget Catalog: TimePicker**

**Layout:**

```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"
    android:gravity="center_vertical">
    <TimePicker
        android:id="@+id/picker"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>
</LinearLayout>
```

**Activity:**

```java
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();
    }
}
```
Figure 682: Android 2.3.3
Figure 683: Android 4.0.3
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:
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
<?xml version="1.0" encoding="utf-8"?>
<!-- WidgetCatalog: ViewFlipper -->
<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>
                <LinearLayout>
                    <!-- items -->
                </LinearLayout>
            </LinearLayout>
        </ViewFlipper>
    </ViewFlipper>
</LinearLayout>
```

**Activity:**

```java
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) {
        // onCreate
    }
}
```

2590
super.onCreate(icicle);
setContentView(R.layout.main);

flipper=(ViewFlipper)findViewByld(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.
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
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.
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
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

```xml
<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:
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 684: The original Kindle Fire, running a sample application, showing the menu bar](image)

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:
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.
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:
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:
Figure 687: 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:
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.
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.

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.
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
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.
However, Amazon has their own equivalents for some of these:

- The [Amazon Maps API](https://developer.amazon.com/public/solutions/services/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](https://developer.amazon.com/public/solutions/devices/messaging-api) is roughly analogous to GCM for pushing messages to a Kindle Fire device
- The [Amazon In-App Purchasing API](https://developer.amazon.com/public/solutions/in-app-purchasing) 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](https://www.amazon.com/gp/developer) and [Kindle Fire HDX](https://www.amazon.com/gp/developer) 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 StackOverflow](https://stackoverflow.com/questions/tagged/kindle-fire).
Nokia — which, by the time you are reading this, will have been absorbed by Microsoft — has entered into the Android device space with their Nokia X series of phones. On the surface, they are merely inexpensive Android devices. However, Nokia has eschewed the Google proprietary apps, and as a result is carving out their own ecosystem, much as Amazon has for the Kindle Fire series and Fire TV.

Nokia will be releasing several devices as part of the Nokia X series. The initial device is the Nokia X itself, with X+ and XL variants that offer better hardware specifications.
This chapter will focus less on the hardware (which is unremarkable) and more on that ecosystem and how it impacts you as an app developer.

**Prerequisites**

Understanding this chapter requires that you have read the core chapters of this book.

**What Is Really Different?**

What will leap out at you at a glance is the home screen, which is an approximation of the main screen of a Windows/Windows Phone device (a.k.a., “Fastlane”). However, overall navigation is a bit different than what you would expect on an ordinary Android device.

The Nokia X has one off-screen button. A simple tap on it functions as the BACK button, while a long-tap on it functions as the HOME button. On the home screen, a horizontal swipe will move you from the roster of tiles to the recent-tasks list, replacing the long-tap on the HOME button.
Otherwise, navigation is reminiscent of more conventional Android devices, including a vertical bezel swipe to open up the notification drawer. Your action bars will use the embedded “…” for the overflow, as the device lacks a dedicated off-screen MENU button.

Beyond that, the biggest differences are the pre-installed apps. As with Amazon’s Kindle Fire and Fire TV devices, the Nokia X series lacks Google's proprietary apps, or any libraries that are part of those apps. Users will lack Google Maps, Hangouts, Gmail, the Play Store, and so on. Developers will lack the Play Services Framework, which in turn means that developers will lack Google Maps, GCM, LocationClient, and anything else that depends upon Play Services.

For developers, Nokia has some replacements for Play Services APIs, which will be covered later in this chapter.

Note that the X series is currently using a Nokia-flavored version of Android based on 4.1 (API Level 16).

Getting Your Development Environment Established

The Nokia X Developer’s Library is Nokia’s home for getting going with Nokia X series development. Note that the site may not work well with all browsers.

Nokia is distributing an SDK add-on that supplies a variety of X-specific things to Android developers, including an emulator image. To install it:

• Start the SDK Manager, such as via the toolbar button in 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 http://tools.nokia.com/nokia-x/repository/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

At this point, you should have:

• a “Nokia X System Image” emulator image in the API 16 group
You can install whichever of those you feel that you need. The system image and device definitions are for emulators, the “Nokia X services” one is for creating projects that use Nokia’s HERE services, and the USB driver is for Windows developers looking to test on Nokia X hardware.

**Emulator Configuration**

If you installed the “Nokia X System Image” and “Nokia X Device Definitions” as described above, you will be able to set up a Nokia X emulator:

![Figure 691: Nokia X Emulator Configuration](image)

The keys are the “Nokia X” device and the “Nokia X services” target. Emulator images are available in both x86 and ARM variants.

Assuming that your development machine supports it, checking “Use Host GPU” will allow you to use Nokia’s maps in your app on the emulator.
The Nokia X emulator is a fairly faithful representation of what a Nokia X looks like, including the tile-based home screen:

![Figure 692: Nokia X Emulator](image)

**Developing on Hardware**

Nokia has [instructions](#) for getting the Nokia X to work as a development device. In addition to normal stuff — like installing the Windows USB driver if needed and enabling debugging on the device in its Settings app — the instructions will have you add 0x0421 to the `adb_usb.ini` file for your environment (e.g., in `~/.android` on Linux).

**Testing Your App**

As with most environments, testing your app on the emulator helps, and testing your app on the actual device is best. However, Nokia has added a pair of services that can help you with testing your app beyond the standard emulator and buy-your-own device options: a compatibility check tool, and a remote test lab.
Nokia X Analyzer

The Nokia X Analyzer will examine an APK file and tell you about any obvious problems with it, such as reliance upon Play Services. This tool is available online and offline. The offline edition is part of the Nokia X Eclipse Extensions, which also give you access to a code samples library published by Nokia.

The online edition allows you to simply drag-and-drop an APK file onto the page, which will trigger an upload of that APK to Nokia’s servers, which will then produce a compatibility report. That report may point out known issues, like being signed by the debug signing key:

```
INCOMPATIBLE

AppCompatDemo-debug.apk
Size: 692.39 KB

1 Incompatibility detected

Platform

Warnings in manifest

Incompatibility
Incompatible

Alternative
Sign the app with publishing key

Figure 693: Nokia X Analyzer, Benign Results
```

Or, it may detect and report other problems that would prevent you from distributing to the Nokia X, such as using Google's Maps V2:
The offline version has two main benefits:

1. It performs source-level analysis, and thereby can give you more focused assistance, compared with simply examining automatically reverse-engineered copies of your manifest and such from an APK.
2. It avoids uploading your APK to Nokia’s servers, particularly since they claim the right to hold onto that APK and use it over time for their own purposes.

However, it does require you to add Eclipse extensions. Eclipse extensions have a tendency to exacerbate Eclipse’s stability issues, and you may not be using Eclipse as your IDE.

In general, though, the offline edition fills the same role as does the online service, to let you know where your apps will fail to work with the Nokia Remote Test Lab

Nokia also offers their Remote Device Access service, where you can access Nokia devices, including the Nokia X, virtually through your browser. This includes the
ability to install applications, transfer files, and examine LogCat. It does require joining the Nokia Developer network, though, and like all remote test labs, certain capabilities (e.g., camera) may not be readily testable this way.

**Your Play Services Alternative is HERE (In Part)**

Nokia offers three partial replacements for what you lose from Play Services.

- HERE Maps, offering similar mapping capabilities to [Maps V2](https://commonsware.com)
- Nokia Notifications, filling the role of [Google Cloud Messaging](https://commonsware.com)
- Nokia In-App Payments, as a replacement for Google's in-app purchases system

In all cases, Nokia provides guides for helping you port your code from Google's APIs to Nokia's, often times providing sample code or “wrappers” that help make it easier for you to support both APIs from a single APK.

**How Does Distribution Work?**

As with other non-Google-ecosystem devices, users will not be using the Play Store to get your apps onto their Nokia X devices. Instead, Nokia is providing other options, just as Amazon does with the Amazon AppStore for Android for the Kindle Fire series and Fire TV.

**Nokia Store**

Nokia's store is simply called the “Nokia Store”, identified by a simple “Store” tile on the home screen. As with the Amazon AppStore for Android, the Nokia Store is a curated market, where apps will go through a review process (which “takes three to four business days”) before they will be published for users. Beyond that, the submission process is not significantly different from that of other Android distribution channels.

The Nokia Store has its own in-app payment system should you elect to go that route for “monetizing” your apps.

Note that if you use `market:// Uri` values in your app to link to your app on the Play Store, those same Uri values should be supported on the Nokia X series to lead

---

Subscribe to updates at https://commonsware.com  
Special Creative Commons BY-NC-SA 4.0 License Edition
to the Nokia Store. Hence, you should not need to change your code for this to work.

Alternatives

Users can enable the ability to install apps from third-party sources in Settings, and so they can install any publicly-available app distribution client, such as the Amazon AppStore for Android itself. They should also be able to install apps from the Web, in addition to “sideloading” them through adb from the Android SDK tools.

Getting Help

Nokia has indicated that they are monitoring the nokiax tag on StackOverflow. They also have a few message boards for the Nokia X, a portion of their community Wiki set aside for the Nokia X, and some blog posts with the nokiax tag.
While Amazon’s Kindle Fire is all the rage in the 7” Android-based tablet space, it certainly was not the first such tablet.

It was not even the first such tablet sold by a firm known originally for selling printed books.

Barnes & Noble, a large American bookstore chain, released the NOOK Color in November 2010 and followed that up with the NOOK Tablet in November 2011. Like the Kindle Fire, the NOOK series are based on Android but have a substantially replaced home screen and other built-in apps. Also, like the Kindle Fire, the NOOK series eschews the Play Store (and any other Google apps) in favor of its own distribution channel.

This chapter will explore developing for the NOOK series, focusing on the newer NOOK Tablet.

**Prerequisites**

Understanding this chapter requires that you have read the core chapters of this book.
What Features and Configurations Does It Use?

In some respects, the NOOK series is closer in spirit to traditional Android devices than is, say, the Kindle Fire. That being said, there are certainly departures from what you would expect, in many cases to keep the parts count and price low.

Screen Size and Density

Both the NOOK Color and the NOOK Tablet have 1024x600 displays, categorized correctly as -mdpi from a screen density standpoint. The NOOK Color is correctly categorized as a -large screen, given its 7” diagonal display.

The NOOK Tablet, on the other hand, claims to be -xlarge, despite the fact that it too has a 7” diagonal display. This will be a problem if your -xlarge resources (e.g., layouts) are really designed for 10” tablets and will not work especially well on a 7” tablet.

Hardware Features

The NOOK series does not support:

1. Any form of location tracking via LocationManager
2. Recording via a microphone
3. Anything involving a camera
4. Anything involving Bluetooth
5. Gyroscope sensors

In addition, the NOOK devices are not phones and so lack any telephony capability, including SMS/MMS.

What Is Really Different?

Beyond the mis-categorizing of the NOOK Tablet as an -xlarge device, there are other places where the NOOK series has departed from standard Android conventions, even within the flexibility supported by the Android OS.
Status/System Bar and Navigation Norms

If you play around with a NOOK Tablet, you will discover that there are no obvious BACK and MENU buttons anywhere on the screen. Most of the built-in applications eschew BACK and MENU, though, preferring iOS-style on-screen backwards navigation, albeit with inconsistent styling.

If you try your own apps, they should cause BACK and MENU buttons to appear, very small, in the status bar that exists at the bottom of the screen. Most of the time, this status bar simply shows the time, battery charge, etc.

Similarly, there is no HOME button. The raised “horseshoe” button towards the bottom of the device, when pressed, brings up a menu of places to navigate to, one of which is the home screen. Note that this behavior only appears on hardware; the NOOK Tablet emulator seems to completely ignore that button and offers no obvious means of getting back to the home screen directly from your app.

Nothing Googly

As with the Kindle Fire, the NOOK series of devices lack any of the Google apps. This includes Google Play for installing other apps, Google Maps (and the Maps SDK add-on), and so on. You will need to find alternatives as needed.

No Side-loading

As will be discussed in greater detail later in this chapter, side-loading of apps is limited at best on the NOOK series.

Toasts to the Top

Whereas in standard Android, the default positioning of a Toast is towards the bottom of the screen, on the NOOK Tablet, it is positioned towards the top.

Unsupported APIs

The NOOK devices do not support home screen app widgets or text-to-speech.
Getting Your Development Environment Established

The NOOK Developer site offers its own SDK for NOOK development. The NOOK SDK is an add-on for the Android SDK environment, so you will need to start with the standard Android SDK tools and such before proceeding.

From the Android SDK Manager, choose Tools | Manage Add-on Sites... This will bring up a dialog box that you can use for adding vendor-supplied add-ons that are not part of Google's central add-on registry:

![Figure 695: The SDK Manager Add-on Sites dialog](image)

Note that you might need to resize the dialog for the buttons on the right to appear.

In that dialog, click New... and fill in http://su.barnesandnoble.com/nook/sdk/addon.xml as the URL. Then click OK to close the dialog, and Close to close the Add-on Sites dialog. You should find a new NOOKcolor entry in the Android 2.2 section of your SDK Manager, which you can then check and install. You can repeat the process with http://su.barnesandnoble.com/nook/sdk/Nook_Tablet_addon.xml for installing the NOOK Tablet add-on, which will appear in the 2.3.3 section of your SDK Manager.
Emulator Configuration

The primary reason for installing those SDK add-ons is to get access to official NOOK emulator images. With the add-ons installed, your AVD Manager will allow you to create NOOK Color and NOOK Tablet emulators, just as you would create emulators for various Android API levels.

Note that the NOOK emulator has a lot of space-consuming chrome around the actual display:

![Figure 696: The NOOK Tablet emulator](image)

As a result, you may need to scale the emulator down smaller than the physical 7” of the actual device, simply because the emulator image is too tall.

Also note that the NOOK emulator does not correctly report an OS version to Eclipse, so you may find that when you try to run your app, Android launches some other emulator. Right click on the project and choose Run As > Run Configurations, and change the project to Manual deployment target selection on the Target tab of the Run Configurations dialog.
Developing on Hardware

The NOOK people make it very annoying to attempt to develop on NOOK hardware, for unknown reasons.

Officially, you need to file paperwork to become a “qualified NOOK App Developer”. This is not possible except for US residents (or firms with a US tax ID from a small list of other supported countries). You also already have to have a production app released elsewhere, and your request has to be approved by the NOOK team.

While there used to be procedures for getting past this restriction, recent firmware updates for the NOOK Tablet have blocked those procedures. At the time of this writing, short of fully rooting the device (and potentially replacing the firmware), development on the NOOK Tablet does not appear possible short of going through the official mechanism.

How Does Distribution Work?

Short of rooting and modding, app distribution for the NOOK series is purely through the Barnes & Noble Storefront. There is no fee to become a “qualified NOOK App Developer” to submit your apps to the Storefront, and you get 70% of the list price of paid apps, on par with similar distribution mechanisms.

The limited distribution options for the NOOK series make it an unsuitable device for use with private apps (e.g., enterprise development), short of rooting and modding.
Device Catalog: RIM Blackberry Playbook

Research In Motion (RIM) are known around the world for their Blackberry line of phones and messaging services. In 2011, they leapt into the tablet arena with the Blackberry Playbook. The 2.0 version of the Playbook OS supports running carefully repackaged Android applications, and you can distribute these applications through a RIM-supplied marketplace if you so choose.

**What Features and Configurations Does It Use?**

Android offers a reasonable amount of flexibility to device manufacturers, while simultaneously allowing developers to dynamically adapt to different device capabilities. This section outlines what you should expect from the Playbook.

**Screen Size and Density**

The Playbook has a 7” 1024x600 screen. It correctly advertises itself as a -large-mdpi device and will try to pull its resources from those sets.

**Hardware Features**

The Playbook, like most tablets, is not a phone, and so it does not support any telephony capability, including SMS/MMS.

Beyond that, Android apps on the Playbook cannot access:

1. Some sensors, notably proximity, ambient light, and barometer
2. Bluetooth
3. miscellaneous other ill-supported technologies (e.g., NFC)

Also, like most tablets, the Playbook does not have any sort of navigation input besides the touchscreen — no D-pad, trackball, arrow keys, etc. Hence, if you have <uses-configuration> elements that require one of these, your app will not work on the Playbook.

What Is Really Different?

The Playbook is different in part because it is not truly an Android device, but a Blackberry device that happens to have a “runtime” for Android, much like a Web browser might have a runtime for Flash. As such, there are going to be a number of things that will depart from the Android norm that your application might expect.

Navigation

Like most Android tablets, the Playbook offers 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. Users can also learn the BACK gesture – a diagonal swipe from southeast to northwest.

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.

Nothing Googly

By definition, a non-standard Android device lacks the Google apps, such as Google Play, Google Maps, and so on. The Playbook does support geo: as a scheme for an Intent when used with startActivity(), but you cannot directly integrate Google Maps into your application using MapsView and MapActivity. RIM recommends using WebView and the Google Maps Web-based APIs instead.

BARs as Packages

One of the biggest differences, compared to other Android-based devices, is the application file format. You are used to distributing APK files, whether via Google Play or by other means. The Playbook, instead, plays BAR files. You will need to go
through a process to “repackage” your APK into a BAR file for local testing or uploading to Blackberry App World. Fortunately, RIM provides a number of means for doing this, described later in this chapter.

Unsupported APIs

In addition to the Google app limitations, the Playbook does not support:

1. Home screen app widgets
2. Any app with more than one launcher activity
3. SIP
4. Native code via the NDK
5. Text-to-speech
6. Task management APIs, notably those protected by GET_TASKS and KILL_BACKGROUND PROCESSES
7. Some methods on AudioManager and MediaPlayer, mostly targeting Bluetooth devices and vibration motors
8. The Camera class (though accessing the camera via ACTION_GET_CONTENT should work)

In addition, the Playbook does not support some media types normally supported by Android, including Ogg Vorbis, AMR, FLAC, MIDI, H.263, and VP8.

Package Name Length

The Playbook Android runtime only supports package names of 29 characters or less. The build tools will truncate your package name as needed, though you may need to give it some assistance to determine how best to do that (e.g., use the first 29 characters? the last 29 characters?).

Getting Your Development Environment Established

Developing for the Playbook is significantly different than is developing for other Android devices, simply because the Playbook is not really an Android device. It is a Blackberry device that happens to have an Android runtime environment in it.
Checking and Repackaging Your App

You need to convert your APK into a BAR file for test it on the Playbook Simulator or on an actual Playbook. There are three ways to go about this: use an Eclipse plug-in, use an online packager, or use some command-line tools.

**Eclipse Plugin**

RIM 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 and OS X; RIM does not mention support for Linux. This plugin also supports the creation of run configurations to deploy your app to a device or simulator, including supporting IP-based debugging.

**Online Repackager**

For lightweight use, RIM supplies a Web-based version of the same tools, minus the Eclipse integration and debugging. This too, though, only supports Windows and OS X, despite being browser-based. It relies on a Java applet, so your browser will need to have that enabled as well.

**Command-Line Tools**

The only option available for Linux are the command-line tools (though these also support Windows and OS X). There are separate commands for the major steps in the process:

1. `apk2barVerifier` runs a validation check to see if you obviously use or do something that makes your app incompatible (e.g., require API Level 11, as the Playbook runs API Level 10)
2. `apk2bar` creates a BAR file out of the APK file (optionally running `apk2barVerifier` first, to save you running that separately)
3. `batchbar-deploy` will upload one or more BAR files to a device or running copy of the simulator
4. etc.

**Playbook Simulator**

RIM offers a Playbook simulator in the form of a VMWare image. Because of the nature of their Android runtime for the Playbook, RIM does not support the
standard Android emulation environment. And, given the extensive modifications to their edition of Android, you are probably better served either trying to use their VMWare image or developing on actual Playbook hardware. This image is certified for use on Windows and OS X, though RIM does not mention support for Linux (even though there is a VMWare player for Linux).

The VMWare image will have its own IP address, which you can obtain from the Playbook 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**

The Playbook 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. Signing credentials are available from RIM through a Web form, though they require agreeing to a fairly lengthy SDK License Agreement.

**How Does Distribution Work?**

RIM is expecting apps to be distributed to the Playbook primarily through their App World site, which also has Playbook apps that are native to the device (vs. running in the Android runtime).

**Blackberry App World**

Compared to marketplaces for apps for some non-standard Android devices, Blackberry App World is full-featured and extensive. It not only supports the Playbook but all app-capable Blackberry devices. At the present time, App World does not take a percentage of each sale.

**Alternatives**

Side-loading is possible, using the techniques from development. However, there is no indication that over-the-air installation is possible other than through Blackberry App World.
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.

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 is still sitting on the technology from WIMM Labs that, in many ways, presaged current interest in wrist wearables.
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.

**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 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.

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.

**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.

**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.
WRIST WEARABLES

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.

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, at 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.

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](http://www.sony-smartwatch.com) 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.

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.
WRIST WEARABLES

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.

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 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.

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.

This does not prohibit the creation of secure wearable apps, but it make it more difficult.

**What About Android Wear?**

[Android Wear](https://commonsware.com) is an initiative by Google for wearable devices, with a particular emphasis on wrist wearables. It will include a developer SDK, presently available in a preview release.

This book will cover Android Wear once Wear-capable devices exist.
As Android increases in popularity, we are seeing a few devices (or device categories) that become popular but are outside the mainstream, defined here as phones and tablets that legitimately have the Play Store on them. Of course, there are lots of devices that fall outside the mainstream that are not popular — most likely, you do not care about these unless you have a specific need to have your app run on one (e.g., particular device bought by your firm for its field staff). But the devices profiled in this part of the book are popular enough that you might want to consider addressing them, despite the additional “fragmentation” they introduce.

The first such device category is Google TV. At the time of this writing, it has been about 18 months since Google TV was announced (at the 2010 Google I/O conference) and over a year since devices started shipping. However, only recently have we been able to create apps for Google TV devices, let alone users be able to install them. This chapter outlines what you will need to consider if you want your apps to be on Google TV... or perhaps if you do not want your apps to be on Google TV.

At the time of this writing, Google TV runs Android 3.1. Hence, it supports things like fragments natively, without necessarily having the need for the Android Support package. Of course, you may be using the Android Support package for other devices (e.g., Android 2.x phones), and that works perfectly fine on Google TV.

**Prerequisites**

Understanding this chapter requires that you have read the core chapters of this book.
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 Google TV or arrange to not be available for Google TV on the Market.

Screen Size and Density

Google TV devices are always categorized as large screen size. Hence, you will tend to put your layouts in res/layout-large/, or possibly res/layout-large-land/ (since Google TV presumably will always consider itself to be landscape).

Densities, however, are a bit more complicated.

Google TV is for use with HDTV, whether Google 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 will be categorized as an xhdpi density device. A 720p television will be categorized as a tvdpi device, where tvdpi was a new resource set qualifier added in API Level 13. 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

Google TV is not considered to be a touchscreen device. As such, from a resource standpoint, you can use -notouch to isolate resources that should be used on Google TV (or, potentially, other future non-touchscreen devices, should they arise). Hence, if you want a different UI for Google TV than a tablet — to address navigational differences, for example — you can use res/layout-large-land-notouch for Google TV and res/layout-large/ and res/layout-large-land/ for other types of large-screen devices.

Other Hardware

Google TV has no sensors, no camera, no Bluetooth, no microphone, and no telephony features. As such, any application requiring such features will not run on Google TV and will not even show up in the Play Store for such devices. The Google
TV developer documentation has the roster of specific <uses-feature> names that must not be referenced.

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 Google 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.

The Emulator

The Google TV add-on for the Android SDK offers an emulator. However, it does not work like the emulator for standard Android. Instead of using a qemu-based emulator, the Google TV emulator uses KVM, a virtualization environment used by Linux servers. While you can get KVM to run on a Linux desktop — perhaps with some tweaking — it is not available for Windows or OS X development machines. Moreover, KVM cannot itself run in a virtualized environment, so you cannot use VirtualBox or similar solutions to have a Windows or OS X machine run a copy of Linux that, in turn, would run a copy of the Google TV emulator.

For Linux developers, the headaches are modest. For Windows and OS X developers, the options are far from ideal:

1. Use a spare PC that you happen to have lying around for a Linux environment, bearing in mind that not all CPUs and BIOSes support the virtualization extensions required by KVM
2. Attempt to create a bootable USB key that contains Linux and the Android SDK with the emulator, so you can test your app on your existing PC
3. Buy a Google TV device and test exclusively on hardware (downside: unless you have two televisions, you will not be able to test both 720p and 1080p display sizes)
4. Switch to Linux for your development needs
CPU and NDK

Google TV devices will be built on both ARM and Intel chipsets. The devices that shipped in 2010 and most of 2011 were based on Intel Atom chips, but future Google TV devices may use ARM as well.

This should not matter much for you right now, simply because you cannot use the Native Development Kit (NDK) with Google TV at this time. With luck, support for this will be added in the not-too-distant future, particularly for game development.

Overscan

Since Google 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 Google TV devices will generally be connected to the Internet, it may not be via WiFi. Since Google TV devices generally are not portable, some will have Ethernet jacks, and hence some users will elect to wire in their Google TV as opposed to using WiFi.

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, Google TV devices will tend not to move, earthquakes and large dogs notwithstanding.

As such, Google 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 Google TV device will be ineffective.

You can get the approximate location of a Google TV device by using the "static" location provider (e.g., getLastKnownLocation("static")). Unfortunately, there is no SDK-defined static data member for "static" at this time.
However, since users of Google 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**

Handheld Android devices have few buttons, with the number of buttons decreasing as time goes along. The only ones related to media are volume rockers, and perhaps a CAMERA button.

Google TV devices will be manipulated by remote controls. Many of these remotes will have lots and lots of buttons, akin to the remotes people are already used to. In addition to perhaps having a QWERTY keyboard, these remotes will have 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 Google TV application may wish to respond to these, via onKeyDown() in a View or Activity. In particular, a Google 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.

**Channels and Listings**

Unlike most handheld Android devices, Google TV is optimized to accompany some sort of television signal, whether that be cable, satellite, over-the-air HDTV, or something else. Not surprisingly, Google TV offers some TV-specific capabilities that you can elect to employ if it makes sense for your app.

Google TV has a ContentProvider for the device’s channel lineup, so you can present a list of the available channels in a ListView, Spinner, etc. You can query on content://com.google.android.tv.provider/channel_listing and get back columns like the channel_name and channel_number. Note that you will need to hold the com.google.android.tv.permission.READ_CHANNELS permission for this to work.
Another column you can retrieve from the ContentProvider is channel_uri. This is a Uri within that ContentProvider, representing a specific channel. You can create an ACTION_VIEW Intent on that Uri and call startActivity() on it to switch to live TV and change the channel to that channel. This requires sufficient integration between the user’s Google TV device and the source of the signal (e.g., using an “IR blaster” to control an external cable box to change channels), and so this may not work for all users.

**User Base**

As Android has evolved, so has the way its devices get used. Phones are still frequently considered to be very personal, private devices. However, tablets are becoming more shared — witness the XOOM Family Edition from Motorola Mobility. Televisions, of course, are also usually shared among household members. However, Android does not have a system-wide concept of separate user environments, though there are some enterprise add-ons that are making inroads in this area.

Depending on the nature of your app, you may wish to consider setting up your own concept of separate “accounts” for different users of the same device, so they can keep their content and settings separate. If needed, you might consider adding authentication of one form or another, to minimize the odds of one person getting into another person’s stuff. While Android 4.2 has this sort of per-user storage built in, Google TV is running Android 3.1 at the time of this writing, and so 4.2 capabilities are unavailable to you.

**Getting Your Development Environment Established**

If you want to develop for Google TV, you will need to do a bit of work to extend your development environment, as is outlined in this section.

**Installing the SDK Add-On**

In the Android SDK Manager, in the “Android 3.1 (API 12)” section, you will find a “Google TV Addon by Google Inc.” entry, which you will need to install:
**Getting KVM Set Up**

Details of installing KVM will vary by Linux distro. For example, to install KVM on Ubuntu 10.04 or later, you would need to install a few packages:

```bash
sudo apt-get install qemu-kvm libvirt-bin bridge-utils
```

You will also need to log out and log back in, so a change in your account's group membership takes effect.

Full Ubuntu KVM installation instructions can be found on the [Ubuntu Web site](https) — similar instructions are (hopefully) available for whatever distro you are running.

Note that tools related to managing KVM virtual machines (e.g., Ubuntu's `ubuntu-vm-builder` and `virt-viewer` packages) may not be needed, as the Android SDK will be creating your virtual machines for you.

**Creating the Emulator**

You will want to create four emulator images. Both should specify "Google TV Addon (Google Inc.) - API Level 12" as the target. The difference between the four will be their skins, dictating their resolutions:

1. 1080p
2. 1080p-overscan (which simulates the loss of available pixels due to overscan effects)
The rest of the setup should be as normal for your preferred emulator options (e.g., an SD card sufficiently large to hold any test media for external storage).

**Connecting to Physical Devices**

Normally, when developing using Android hardware, you connect your development machine to the hardware via USB. This is not supported by Google TV, perhaps with an eye towards not requiring Google TV-powered televisions to sport USB ports. Instead, you develop for Google TV by TCP/IP. The tools are mostly ignorant of the difference – only `adb` knows and cares about the USB versus TCP/IP differences.

First, you will need the IP address of your Google TV device. You can get this via All Apps > Settings > Network > Status — “IP Address” will be one of the listed pieces of information:

![Figure 699: IP Address on Google TV](image)

With luck, your network will keep the same IP address assigned to this device, even if you shut down or reboot the device from time to time.
Then, at the command line, run `adb connect`, supplying the IP address of the Google TV device. If the `adb` command is not in your development machine’s PATH, you will find it in the `platform-tools/` directory of wherever your SDK is installed. At this point, DDMS and `adb devices` should report the Google TV device. Rather than the device ID being a serial number or `emulator-5554`, it will be the IP address plus :5555.

At this point, all your normal tools should work, for viewing LogCat and so on. The screenshot shown above, for example, was taken using the DDMS perspective in Eclipse. Note, though, that the screenshots will only be from what Google TV is generating, not any underlying picture being supplied by your television signal input.

To disconnect, simply run `adb disconnect` with the IP address of the Google TV device.

**How Does Distribution Work?**

Your app probably falls in one of three buckets: you want it on Google TV (along with other devices), it *only* supports Google TV, or it will not work on Google TV. Whichever of those buckets best fits your device will determine the manifest 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 Google TV**

The first criterion for getting your app visible to Google 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:

```xml
<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 Google TV may lack. The [Google TV developer documentation](https://commonsware.com) has the full roster of unsupported features.
Also:

1. If you have any `<uses-configuration>` elements in the manifest, double-check to make sure that they will be possible on Google TV devices. The configurations that Google TV does not support are ones where you need the touchscreen (android:reqTouchScreen="stylus" or "finger").
2. Do not have any activities with android:screenOrientation set to portrait, as Google TV devices always display in landscape.
3. Apparently not all OpenGL textures are supported, so if you are using `<supports-gl-texture>` elements in your manifest, you will need to ensure that such textures work on Google TV, presumably via testing.

**Supporting Only Google TV**

If your app only supports Google TV, in addition to the above requirements, you should also add one more `<uses-feature>` element to your manifest:

```xml
<uses-feature android:name="com.google.android.tv" android:required="true"/>
```

This will filter you out of the Market for all non-Google TV environments.

**Avoiding Google TV**

If your app specifically is untested on Google TV, you need to have something in the manifest that will keep you off Google TV devices’ views of the Play Store. The easiest is to say that you need a touchscreen:

```xml
<uses-feature android:name="android.hardware.touchscreen" android:required="true"/>
```

**Dealing with Other Televisions**

There are other devices that support Android on televisions. While few of these exist as of the summer of 2012, many are in the works, such as the oft-cited, crowd-funded OUYA console.

Android 4.1 (a.k.a., Jelly Bean) added a separate feature for televisions: android.hardware.type.television. Requiring this would limit your application to devices that are to be displayed on televisions.
DEVICE CATALOG: GOOGLE TV

However, as of the time of this writing, it is unclear which, if any, devices or markets honor this particular <uses-feature> element.

Getting Help

The Google TV Developer site has a lot of information on creating Google TV apps, in terms of design and implementation details.

The primary place to get your questions answered regarding Google TV development is StackOverflow’s google-tv tag.
Amazon has joined the TV set-top box fray with the Fire TV. This device is 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.

This chapter will review the Fire TV itself, from a developer's standpoint, to help you create apps for this platform. Note that the Fire TV has 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 Google TV as a point of reference.

**Introducing the Fire TV**

As with most Android-powered set-top boxes, Fire TV is small and is designed to connect with your television (or monitor, or projector, or whatever) via HDMI:
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:
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:
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):
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:
What Features and Configurations Does It Use?

The Fire TV behaves a bit like a “mashup” of other TV-centric devices (Google TV, OUYA, etc.) and the Kindle Fire series.

**OS Version**

At the present time, Fire TV is powered by a version of FireOS that corresponds to Android 4.2.2 (API Level 17). It is possible that Fire TV will 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 Fire TV differs from pretty much anything that came before it comes with the behavior of the screen. As with other TV-centric devices like Google TV and the OUYA, the assumption is that the screen is locked to landscape. However, beyond that, Fire TV has struck out on its own in terms of screen characteristics.
With Google TV, the screen size and density are based on the type of display that the Google TV box is plugged into. You work with the actual available screen space, less any space that Google TV sets aside for dealing with overscan.

With Fire TV, your code deals with a rendering surface of 1920x1080 pixels, regardless of whether the Fire TV is plugged into a 1080p screen or something else (720p, 4K, etc.). The density is always treated as -hdpi, giving you a resolution of 960dp by 540dp, and a large screen size. Fire TV will handle scaling the output down (or, in theory, up for 4K) as needed, plus addressing any overscan issues.

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, since Fire TV’s overscan support is not documented, you may be well-served by avoiding putting important content towards the screen edges.

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, be sure to allow your activities to work in portrait or landscape, not just portrait.

Input Devices

As noted earlier, the Fire TV ships 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 is not a touchscreen device. 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, the Fire TV lacks a lot of hardware that you normally associate with phones and tablets, such as:

- GPS
• camera
• microphone
• sensors
• telephony

Note that while the remote has a microphone, for use with the Fire TV search capability, there is no current evidence that developers can use it officially.

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 Cloud Messaging. Note that the Amazon Maps API does not work on the Fire TV, as the Fire TV has 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.

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.

## Getting Your Development Environment Established

Developing for the Fire TV, as with developing for the Kindle Fire series, is best accomplished using an actual Fire TV device. That being said, having an emulator that at least resembles the Fire TV 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.

To install it:

- Start the SDK Manager, such as via the toolbar button in 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.

In truth, at present, you will need to use Fire TV hardware to develop for Fire TV.

### Developing on Hardware

The primary thing notably different about testing your app on a Fire TV 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 Fire TV, this is in Settings > System > Developer Options:

![Figure 705: Fire TV Developer Settings](image)

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:
You can then run `adb connect [address>` at a command prompt to make the connection, substituting your IP address for `<address>`. 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's IP address, and the Fire TV 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 is powered down. You can use `adb disconnect [ipaddress>` 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.

**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
**Amazon Fire TV**

*Focus management and accessibility.* BACK and the D-pad buttons will work on the Fire TV much as you would expect.

The Fire TV 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.

**Gaming Controller**

The wireless remote’s input options are purely buttons. The Fire TV 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.

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.
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 MotionEvent (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, the Fire TV lacks the Play Store. If you want your app to be available to Fire TV 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 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.

“Sideload” 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

Amazon maintains a set of documentation related to Fire TV development, along with a set of forums for asking Amazon-specific development questions regarding the Fire TV or their various SDKs.
Trail: Appendices
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-adaptor**

The cwac-adaptor repository contains a small AdapterWrapper class that wraps aListAdapter. The default implementation of allListAdapter methods is to forward the request along to the wrappedListAdapter. 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.
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-merge

The cwac-merge repository contains MergeAdapter. It simply stitches together lots of smaller ListAdapters into one largerListAdapter 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.

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 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.

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.